Computer Science

Capstone Report - Spring 2022

Evaluating Machine Learning Approach in the Context of Automatic Test Oracle Generation

Harry Lee

supervised by
Lihua Xu

Preface

This study investigates the application of machine learning in the field of software testing, in particular, automatic test oracle generation. AI test oracle generation shows great potential in assisting human testers in the software testing process, which improves the efficiency of software development. In this paper, we will propose a machine learning pipeline, along with several active learning strategies. Then, based on our proposed pipeline, we evaluated several machine learning algorithms, along with relevant strategies. Software practitioners will be able to follow our proposed pipeline, and evaluate different machine learning algorithms with our baseline.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Prof. Lihua Xu for her guidance and assistance over the past two years. Being in her software testing research team is the most valuable experience of my college career. I would also like to express my appreciation to Qingshun Wang and Xin Gong, who provided me with valuable help and suggestions over the entire project. I would also express my gratitude to Prof. Olivier Marin for his guidance and encouragement for my computer science senior project, and to all NYU Shanghai computer science professors for their instructions and care over my college years. Finally, I would like to thank my parents and friends for their care and support, whom without this would have not been possible.

Abstract

In the field of software testing, test oracle is a mechanism that determines whether software executes correctly with respect to a test case[1]. In industry, human testers often undertake the role of test oracles - they review the output of each test case and decide whether it passes or fails. However, when it comes to fuzzing and some other automatic test generation techniques, the nature of test case redundancy makes it almost impossible for human testers to examine all the outcomes of the test cases.

Machine learning (ML) may enable the automated generation of test oracles. So far, different studies have been working on different ML algorithms in the field of software testing. However, a unified pipeline and baseline are missing that comparing these machine learning strategies comes to be intractable. In this paper, we will propose a machine learning pipeline and several active learning (AL) strategies that integrate the data generation phrase all the way towards the evaluation phrase; we will also evaluate several machine learning algorithms and our proposed active learning strategies using the pipeline. Software practitioners will be able to follow our proposed pipeline, evaluating and comparing different ML and AL strategies.

Keywords

Software testing, test oracle generation, machine learning, active learning, capstone. computer science, NYU Shanghai

Contents

1. Introduction

A key component of software testing is deciding whether a test case has passed or failed. The mechanism that enables us to distinguish passing and failing cases is called a *test oracle*. As the knowledge of expected program behavior is essential, the current practice of industry is usually the *human oracle*. However, testing with human oracle is inefficient and impractical for many scenarios, especially when it comes to fuzzing and some other automatic test generation techniques. Thus, an automatic test oracle becomes helpful and essential.

Machine learning (ML) may enable the automated generation of test oracles. Since 2002, studies have been published in the field of using machine learning for automated testing. Interest in this topic is growing with the emergence of new and more powerful ML approaches, with over half of the studies published since 2016[2]. As a subset of machine learning, active learning (AL) is a type of algorithm that can achieve greater accuracy with fewer training labels if it is allowed to choose the data from which it learns[3]. In the context of software testing, labeling each test case is a highly consuming task for human testers. Thus, by taking advantage of active learning test case selection methods, the work of human testers could be reduced.

So far, different studies have been working on a variety of ML algorithms in the field of software testing. However, a unified pipeline and baseline are missing that comparing these machine learning strategies comes to be intractable. In our project, we produce a detailed machine learning pipeline for automated test oracle generation, which includes selecting a set of software under test (SUT), creating a set of test cases for each SUT, transforming the execution traces to vector embedding, training the test oracle with ML and AL algorithms, and evaluating the performance of the entire process. We also refine several active learning strategies based on existing studies; and evaluate several ML and AL algorithms with our proposed pipeline. Software practitioners will be able to follow our proposed pipeline, and evaluate different machine learning algorithms with the baseline.

2. Related Work

2.1. Testing and Test Oracles

Test oracle is a mechanism that determines whether software executes correctly for a test case[1]. Barr el al.[4] classified test oracles into 3 categories - specified oracle, derived oracle, and implicit oracle.

Specified oracles judge all behavioral aspects of a system for a given formal specification, which are effective in classifying test verdicts (whether a test case passes or fails), but requires heavy human effort in defining a specification.

Implicit oracles refer to the detection of "obvious" faults such as a program crash. The establishment of implicit oracles does not rely on any domain knowledge or specification, and abundant studies have been conducted in this area.

Derived oracles involve artifacts from which a test oracle may be derived — for instance, a previous version of the system, or some human-labeled data. This kind of oracle is what all ML-related automatic testing oracle generation studies we reviewed focus on.

2.2. Machine Learning

Machine learning (ML) may enable the automated generation of test oracles. In the field of ML-enabled test oracle generation, 3 categories of ML techniques are involved - unsupervised learning, semi-supervised learning, and supervised learning.

Unsupervised learning techniques do not require any labeled training data. That is to say, if we apply unsupervised learning techniques to generate a test oracle, no previous human labeling is needed. Some classic unsupervised learning algorithms include k-nearest neighbors algorithm (k-NN)[5], agglomerative and divisive hierarchical clustering[6] and DBSCAN[7].

Supervised learning techniques require all labeled data to be labeled. This technique is most likely to produce accurate results, but is also least applicable in our context.

Semi-supervised learning techniques lies between supervised learning and unsupervised learning, which requires only part of the training data to be labeled. Classic semi-supervised learning algorithms include *self-training*, *co-training*, and *co-EM*[8].

Another hot topic within the field of semi-supervised learning is *active learning*. The key hypothesis of active learning is that if the learning algorithm is allowed to choose the data from which it learns, it will perform better with less training[3]. The difference between active learning and other semi-supervised learning techniques is that, the set of labeled data is dynamic, instead of fixed, an algorithm is able to select a subset of training data to be labeled.

2.3. Using Machine Learning to Generate Test Oracles

Several studies have been working on utilizing machine learning to generate test oracles, including works by Almaghairbe and Roper[9, 8, 10], Geethal[11], Braga et al[12], and Arrieta et al.[13].

To build and test a derived oracle, several common steps involved in this process.

2.3.1. Data Preparation

To generate a test oracle, a set of software under test (SUT) should be prepared, including a buggy version and a fixed version. The buggy version simulates the real SUT in the industry, while the fixed version is used to generate the correct output representing human labeling and the evaluation process. The choices of SUT vary from study to study, some SUTs come from open source software[9, 8, 10], while others are industrial applications[13].

Defects4J[14] is a database of existing faults to enable controlled testing studies for Java programs, which includes a set of SUT and each SUT has several buggy versions. Defects4J has been used in several studies in the field of automatic test case generation and automatic program repair, such as one automatic program repair study conducted by Martinez et al.[15].

2.3.2. Feature Engineering

Feature engineering refers to the process of selecting, manipulating, and transforming raw data into features that can be directly learned in machine learning. In our case, feature engineering involves the data processing of the execution trace. Almaghairbe and Roper[8] present several ways for feature extraction towards input/out pair and execution trace. For complicated input/output data, if they cannot be directly considered as quantitative or categorical data, it will be tokenized and converted to a vector; for execution trace, they used a hash map to convert the trace into vectors.

2.3.3. Machine Learning Algorithms

There are 2 categories of machine learning approaches in our selected papers: unsupervised learning and semi-supervised learning. Due to the huge cost of human labeling and our assumption that automatically generated test oracle could eventually replace part of human work, many studies do not choose the direction of supervised learning, instead, they focus more on unsupervised learning and semi-supervised learning.

Roper [10] argues that after clustering the test cases with unsupervised learning algorithms, failing cases are more likely to appear in the clusters with smaller sizes. They later applied another semi-supervised learning algorithm using only 4% of total data from the smaller clusters, and reaches a classification accuracy of 63.9%, and when using 18.3% of data, the accuracy reaches

85.8%.

Despite classic semi-supervised learning algorithms, Geethal[11] applies an active learning approach to automatically "request" humans to label new data. They utilize LEARN2FIX[16] by taking advantage of its active learning approach, which eventually classifies test cases with 63% - 80% accuracy.

3. Solution

To mimic industrial test scenarios and gain required data for training and testing, figure 1 shows our proposed pipeline. Our pipeline consists of 3 phrases - test generation phrase, machine learning/active learning phrase, and evaluation phrase.

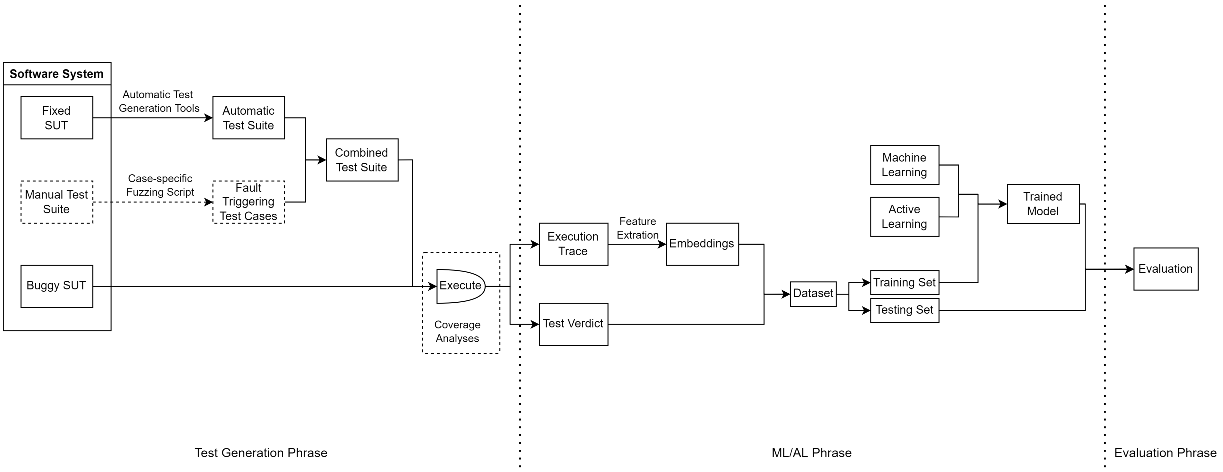


Figure 1: The overall pipeline of the test environment, including a test generation phrase, machine learning/active learning phrase, and evaluation phrase.

3.1. Test Generation Phrase

The goal of the test generation phrase is to produce execution traces of both passing cases and failing cases which maximally explores the SUT. To distinguish passing and failing cases, both versions of the SUT are required - a buggy version and a fixed version. We generate test cases with automatic test generation tools on the fixed version, so that we can ensure the correctness of our test suite. Then, by executing the test suite on the buggy SUT, we are able to see which test cases raise exceptions, which will enable us to output the test verdicts.

There might be 2 scenarios, in which the automatic test generation tools fail to produce a stable test suite, or the automatic test suite cannot reveal any bugs in the buggy SUT. The former case may be due to some random generators or timing functions that bring randomness. To avoid this problem, we need to drop the SUT or exclude the code that introduces randomness. The latter case is led by the limitation of automatic test generation tools. Despite extending the budget on the execution time for automatic test generation tools, we can also look through the manual test cases written by SUT developers, and write case-specific fuzzing scripts. These scripts will exploit the buggy functions or APIs and randomly generate inputs that will maximally diversify the execution trace of test cases that trigger exceptions.

3.2. Machine Learning/Active Learning Phrase

The ML/AL phrase is where the automatic test oracles are trained. In this phrase, 3 parts are worth investigating - feature extraction, machine learning, and active learning. Feature extraction may take the raw execution trace information, which is basically the list of method/block name and number, and convert them into vectors that can be processed by machine learning or active learning algorithms. The dataset is a combination of execution trace and test verdict. It is then divided into the training and testing set. The training set will be utilized by either the (passive) machine learning algorithm, or, in the context of active learning, the algorithm will first query the labels it requires by its test case selection strategy, and then learn the labeled data.

3.3. Evaluation Phrase

Given the trained model and testing set, we are able to evaluate the model. With respect to the quality of the oracle model, we formalizes the evaluation metrics as follows:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$F1\ score = \frac{2 \times (precision \times recall)}{precision + recall}$$

In the context of test oracle evaluation, TP, TN, FN, FP above are defined as below:

- TP: True positive, a failing test result classified as failing test
- FP: False positive, a passing test result classified as failing test
- TN: True negative, a passing test result classified as passing test

- FN: False negative, a failing test result classified as passing test

In addition, the time consumed for training is also considered.

4. Results

In this section, we will follow our proposed pipeline, and conduct a complete evaluation process of several feature extraction methods, machine learning algorithms, and active learning strategies.

4.1. Experiment Setup

Our pipeline includes several parts that require testers to specify the detailed methods applied to the testing environment. We will illustrate the details of our experiments in this section.

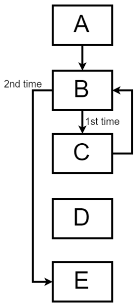
4.1.1. Test Generation

Defects4J[14] has been used in several studies in the field of automatic test case generation and automatic program repair. In our experiment, we choose to utilize this database. When selecting SUT from the Defects4J database, we are mainly concerned about 1) whether the output for each generated input is stable and reproducible, and 2) whether random fuzzing can cover most of the statements of SUT.

We chose Randoop, a feedback-directed regression testing tool for Java programs, to generate our test suite. The test suite generated by Randoop contains only a few assertions statements in each test case, which is able to produce test cases with a unified format while remaining the ability to reach a high code coverage rate. By taking advantage of regression testing, the test suite will include the test oracle by just comparing whether the execution results conform to the results of the fixed version, which makes it easy for us to store the ground truth of test verdicts. Thus, by executing the regression test suite on the fixed version of SUT, we will be able to drop the SUTs that failed to generate stable and reproducible output - this situation often happens to software that takes current time as input, or software that builds on random number generators. However, compared to manual test suite, Randoop suffers from the limitation where the difference between the fixed and buggy version of SUT can be only distinguished through some corner cases. For example, when the condition of triggering an exception is only when input is between $[0.1e, 0.2e]$ or $[0.1E, 0.2E]$, or when the input must follow a certain semantic format - such as XML or JSON. To avoid these scenarios, we have selected a subset of Defects4J programs which only take numbers, or strings without complected format requirements as input. After generating the test suite, we manually examine each of them before collecting the execution trace - if a regression test suite fails to generate test cases that can trigger an exception, we will manually add those triggering conditions into the regression test suite, which are generated by our case-specific random fuzzing scripts that can trigger the exceptions while increasing the variability in their execution traces. Finally, we used our modified version of JaCoCo[17] to collect the program execution trace. Compared to the original version of JaCoCo, our modified version supports collecting the order of program execution, instead of whether a block has been covered or not.

4.1.2. Feature Extraction

To maximize the effectiveness in extracting information from the execution trace, we have applied 3 feature extraction methods: sequence (seq), block coverage (bc1), and block coverage count (bc2). Figure 2 shows a simplified example of data extraction.



Method	Embedding
seq	[[1,0,0,0,0], [0,1,0,0,0], [0,0,1,0,0], [0,1,0,0,0], [0,0,0,0,1]]
bc1	[1,1,1,0,1]
bc2	[1,2,1,0,1]

Figure 2: An example program and its embedding generated by different feature extraction methods, where the order of block execution is $[A \rightarrow B \rightarrow C \rightarrow B \rightarrow E]$. Each node in the graph represents a block of SUT. A block is a piece of code with exactly one entry and exit point. In this example, the order of the one-hot embedding for each block is in alphabetic order.

The common ground for the three methods is one-hot encoding. Each position of the one-hot vector corresponds to a unique block. For example, the one-hot vector for block A is $[1, 0, 0, 0, 0]$.

For the **sequence method (seq)**, we stack the one-hot vectors for each block in their execution order, which forms a 2D matrix - the rows of the matrix correspond to the unique blocks, and the columns correspond to the executed blocks. In addition, to avoid execution trace explosion, and unify sequence length (so that PyTorch can process it with GRU and LSTM), we pad or truncate the sequence at a fixed length by the following equation:

$$length = \min(\alpha, \mu + \sigma)$$

, where α is the maximum length of all execution trace sequences, μ is the mean of all execution trace sequence lengths, and σ is the standard derivation of all execution trace sequence lengths. We assume the length of execution traces follows a normal distribution, and this equation guarantees that most sequence information is fully reserved, while only a few sequences are truncated - but we believe that the parts we keep are sufficient enough for the algorithms to make decisions.

The **block coverage count (bc2) method** records the times that a block is visited. The length of the bc2 vector corresponds to the number of unique blocks. Bc2 vector can be directly computed by summing over the columns of the seq matrix.

The **block coverage (bc1) method** records whether a block has been visited. The length of the bc2 vector corresponds to the number of unique blocks. Bc1 vector can be directly computed by taking the Boolean value of the bc2 vector.

4.1.3. Machine Learning Algorithms and Techniques

In our experiments, 2 types of machine learning methods are explored: statistical learning and deep learning. For statistical learning, we explored logistic regression (LR), support vector classifier (SVC), and random forest (RF) algorithms. For deep learning, we explored 2 types of recursive neural networks (RNN): gated recurrent unit (GRU) and long short-term memory (LSTM).

Statistical learning methods are able to capture information from simpler data. Thus, bc1 and bc2 methods are tested for the 3 algorithms. Among the 3 methods, LR focus directly on predicting the probability of each test case, SVC aims to build a classification boundary between test cases, while RF, as one bagging of decision trees, focuses on classifying test cases feature by feature.

RNN has the ability to cope with sequential data. The advantage of seq method over bc1 and bc2 is that, seq maintains the information of the execution order of each block. However, it also increases the complexity of the input vector. Thus, we utilize GRU and LSTM - 2 different variations of RNN. GRU has a simpler architecture than LSTM, which is preferred in smaller datasets, while LSTM is preferred in larger datasets.

One challenge in our experiments is that, the training data is highly imbalanced. Table 2 shows the number of passing and failing cases in our experiment. To avoid the machine learning algorithm from flavoring the passing cases, we reweight the training data to ensure that the sum of positive weights equals the sum of negative weights.

Project	**Failing Cases**	**Passing Cases**	**Failing Rate**
Lang-1	35	8445	0.41%
Lang-3	6	7833	0.08%
Math-1	4	904	0.44%
Math-3	5	773	0.64%
Math-4	3	444	0.67%
Time-1	25	550	4.35%

Table 2: Passing and failing cases in each test suite and Defects4J project. Failing rate = number of failing cases / number of total cases.

4.1.4. Active Learning Techniques

As a subset of machine learning, active learning is a type of algorithm that can achieve greater accuracy with fewer training labels if it is allowed to choose the data from which it learns[3]. The most important part of conducting active learning is how data is selected from the training set. In our experiments, we adopted and improved the methods proposed by Groce et al.[18], 3 data selection methods are explored: cosine distance prioritization (cos-dist), confidence prioritization (confidence+), and random selection (random).

Cosine distance prioritization (cos-dist) method builds upon the idea of semi-supervised learning. It assumes that similar cases are more likely to have the same label. Thus, instances most distant from the selected set should be tested first. In practice, this method is highly time-consuming since the algorithm requires computing the distance between each point of the selected set and the candidate set (the set of data that is not yet included in the training process). Thus, random sampling is used for approximate results. In practice, we set the random sample size to be 16 times the number of new data to be labeled.

Confidence prioritization (confidence+) method builds upon the fact that all our selected machine learning models are able to generate a prediction of probability for each execution trace. Thus, by flavoring the predicted probability closer to 0.5, we are able to prioritize the test cases that the model feels most "uncertain". However, statistical learning methods require the training set to include both positive and negative labels. Thus, when not all labels are yet discovered, we applied the cos-dist method in practice. On the other hand, we utilize the confidence prioritization method for deep learning models throughout the entire training process.

Random selection (random) method works as a baseline. By comparing the result between random selection and the 2 methods above, we can find out how well the selection methods have been improved.

4.2. Evaluation

4.2.1. Explore SUT with (Passive) Machine Learning

Table 3 shows the evaluation metrics for machine learning algorithms. For LR, SVC, and RF, we applied both bc1 ad bc2 feature extraction methods, and for GRU and LSTM, we only applied the seq method for feature extraction. Our results show that, in the scope of statistical learning methods, random forest (RF) has the best overall performance on its F1 score, which shows its ability in discovering failing cases (high precision), while generating fewer "false alert" (high recall). However, logistic regression (LR) achieves the highest recall, but a lot of its positive predictions turn out to be negative (low precision). And support vector classifier performs the worst among all our tested algorithms. As for deep learning methods, they show the ability to handle complicated data, but their performance shows no superiority over the statistical learning methods. Given that the training time of these models takes thousands of times over statistical learning models, we do not consider them to be a preferable method in practice.

Algorithm	**Extraction Method**	**Recall**	**Precision**	**Accuracy**	**F1-Score**
LR	BC1	0.97	0.15	0.94	0.24
LR	BC2	0.97	0.27	0.93	0.33
SVC	BC1	0.47	0.09	0.76	0.11
SVC	BC2	0.33	0.17	0.82	0.17
RF	BC1	0.68	0.82	0.99	0.73
RF	BC2	0.79	0.72	1.00	0.73
GRU	SEQ	0.81	0.63	0.99	0.66
LSTM	SEQ	0.86	0.70	0.99	0.71

Table 3: Average model performance for each combination of feature extraction method and machine learning algorithm among 6 Defects4J projects.

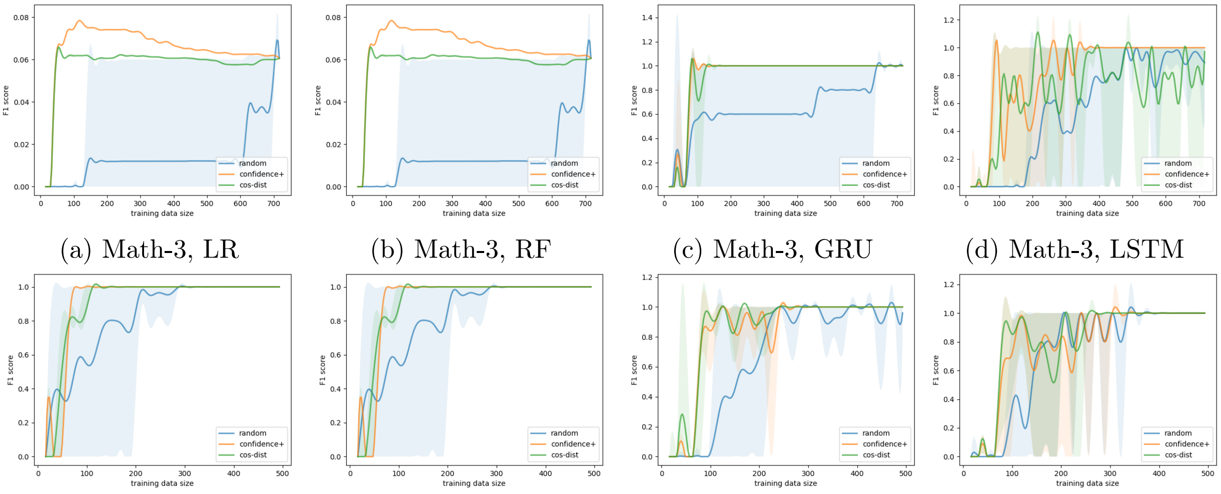
To evaluate the feature extraction methods, we averaged the evaluation metrics over the algorithms, table 4 shows the averaged result. We drop the seq method outside of this table since the results between seq and other 2 methods also relate to the chosen algorithms, which makes it meaningless to conclude which feature extraction method is better. Our result shows that, block coverage count (bc2) outperforms block coverage (bc1) method in precision, accuracy, and F1 score, and their difference in recall are close, which are nearly the same. In theory, bc2 contains all the information that bc1 has, while our results show that statistical learning algorithms are able to utilize this additional information and achieve a better result.

Extraction Method	**Recall**	**Precision**	**Accuracy**	**F1-Score**
BC1	0.71	0.35	0.90	0.36
BC2	0.70	0.39	0.92	0.41

Table 4: Average model performance BC1 and BC2 among 6 Defects4J projects and their applied machine learning results.

4.2.2. Explore SUT with Active Learning

Our experiment with passive machine learning algorithms shows that the SVC algorithm shows a poor ability in distinguishing passing and failing test cases. Thus, in our active learning experiments, we have dropped SVC out of our inventory. Since random selection and the random sampling techniques in cos-dist method bring randomness to the model, for each project, selection method, and machine learning algorithm, we run the same setting for 5 times to eliminate contingency. While taking the mean for evaluating their performance, we also look through the range of each experiment, which shows the stability and variation of each method. In addition, to compare the results across algorithms, we store the data that is randomly generated to ensure that for cos-dist and random methods, each algorithm will be able to fetch the identical training data for each training size. Due to simplicity, figure 3 shows the F1-score with respect to each training set size for 2 of our selected SUT, and the precision and recall curves are provided in appendix A.



(a) Math-3, LR (b) Math-3, RF (c) Math-3, GRU (d) Math-3, LSTM

(e) Math-4, LR (f) Math-4, RF (g) Math-4, GRU (h) Math-4, LSTM

Figure 3: F1-score for projects Math-3 and Math-4, with respect to each machine learning algorithm and active learning technique.

Our results of active learning experiments show that, both confidence+ and cos-dist method outperforms random selection, and they tend to produce a much more stable result. Compare to cos-dist, confidence+ performs slightly better. As confidence+ is more time-efficient than cos-dist, it is considered to be the most preferred method in practice. In addition, compared to statistical models, deep learning model results are more unstable. Although we have increased the number of epochs for training, the result still turns out to be unstable.

5. Discussion

5.1. Challenges

During the entire project, we've encountered 3 challenges: programming language, testing tools, and test efficiency.

Running Defects4J requires basic knowledge of Java programming language, specifically, how to write JUnit test cases, including how to write fuzzing scripts that can randomly generate JUnit test cases, and handle exceptions. In addition, the Defects4J database is built on Perl and bash language, which are also unfamiliar to us. Thus, learning the relevant technology stack and getting ready for the testing environment takes a lot of our time.

The original Defects4J framework does not have the ability to record the program execution trace. And the original version of JaCoCo does not record the order of execution. As a result, with the help of Qingshun Wang, we modified the JaCoCo source code, so that the sequence of program execution traces can be stored.

Calculating cosine distance for every pair of embedding vectors is time-consuming. Thus, we take a random sampling strategy to speed up this process. To increase the usability of distances and unify the randomness across experiments with different algorithms, we introduced a cache that stores the random state and each reusable intermediate result, which effectively saves the running time for our evaluation process.

5.2. Reflection on Related Works

Roper [10] argues that after clustering the test cases with unsupervised learning algorithms, failing cases are more likely to appear in the clusters with smaller sizes. In our cosine distance prioritization active learning strategy, we pushed this idea to the extreme - instead of calculating the distance between each cluster in the unsupervised learning context, we consider each execution trace as an identical cluster. The smaller cluster in Roper's study thus corresponds to the vectors that are farthest from other vectors. The advantage of our approach over Roper's unsupervised learning method is that, our approach does not need to specify how many clusters are needed for unsupervised learning (although not all unsupervised learning methods require specifying the number of clusters in advance). In addition, Roper suggests that human testers should start checking test cases from the smaller clusters, but does not mention the examination order within each cluster. Our approach of cosine distance prioritization solves this problem by sorting the execution traces.

The study of Groce et al.[18] shows the ability of support vector machine algorithms to distinguish passing and failing cases. However, in our experiment, the performance of SVC algorithm works not well, and its performance is far behind other evaluated algorithms, although our experiments conform to their results that confidence prioritization works better than cos-dist prioritization.

5.3. Limitations

There are mainly 2 limitations in our study: the problem with SUT and Randoop, as well as the deep learning model.

In our experiment, we use Randoop to randomly generate a huge set of automatic test cases, which mimics the industrial situation where abundant random tests are executed by fuzzing tools. However, the nature of Randoop suggests that it cannot be applied to SUT that has complicated input - for example, string formats such as JSON and XML. Thus, our approach brings 2 problems - 1) SUT cannot be the ones that require complicated inputs, and 2) Randoop test cases may not reach a high block coverage rate for certain SUT. In industry, human testers often utilize some formatted fuzzing tools - such as FACTS[19] and FinExpert[20], which outperforms Randoop in reaching a higher coverage rate and increases the ability of automatic test suite to expose the buggy codes. In addition, the performance of ML and AL methods may be different with respect to each SUT. Our evaluation results may be limited by the range of chosen SUT and test generation tool.

Another aspect of our experiments falls on the machine learning algorithms. We did some basic turning strategies such as reweighting input labels. However, the deep learning methods may need extra refinements. For instance, RNN models often suffer from gradient vanishing and exploding problems. Thus gradient clipping may be useful to alleviate this problem. However, our RNN models are nearly native prototypes without too much adjustment. Thus, though our experiment shows the ability of deep learning models in processing sequential execution trace data, we have not yet reached the upper limit of their performance.

5.4. Possible Other Approaches

There are 2 other approaches that may be interesting to investigate. One is the extensibility of our automatic oracles, another is new machine learning algorithms that show better capabilities.

Our evaluation metrics focus on the performance of automatic oracle on the current version of SUT. However, an interesting research question would be, what if the SUTs have been changed? This problem may cause a challenge when changes in the number of code blocks in a SUT are introduced. However, combined with mutation testing, the extensibility of the current model can be further investigated.

In section 5.3, we mentioned that RNN models suffer from gradient vanishing and exploding problems. Despite improving the RNN architecture, another approach is to use the transformer model[21]. Instead of flavoring recent inputs, transformer models treat all elements in a sequence equally, which is worth trying.

6. Conclusion

In this paper, we go through the entire process of evaluating machine learning and active learning approaches in the context of automatic test oracle generation. In particular, our contributions are as follows:

- We proposed a complete evaluation pipeline consisting of every detail of the test generation phrase, ML/AL phrase, and evaluation phrase. The pipeline overcomes the problem of reproducing machine learning algorithms in the context of automatic test oracle generation. Practitioners are able to evaluate different aspects of automatic test oracle generation with our pipeline.
- We refined and improved 2 active learning test selection strategies - cosine distance prioritization and confidence prioritization, which can effectively reduce the training set size to reach a better performance.
- We did a complete evaluation with our proposed pipeline, and compared several feature extraction methods, machine learning methods, and active learning test selection methods.

References

[1] A. Memon, I. Banerjee, and A. Nagarajan, "What test oracle should i use for effective gui testing?" in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, 2003, pp. 164–173.

[2] A. Fontes and G. Gay, "Using machine learning to generate test oracles: A systematic literature review," *CoRR*, vol. abs/2107.00906, 2021. [Online]. Available: https://arxiv.org/abs/2107.00906

[3] B. Settles, "Active learning literature survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[5] E. Fix and J. L. Hodges, "Discriminatory analysis - nonparametric discrimination: Consistency properties," *International Statistical Review*, vol. 57, p. 238, 1989.

[6] J. H. Ward, "Hierarchical grouping to optimize an objective function," *Journal of the American Statistical Association*, vol. 58, pp. 236–244, 1963.

[7] R. F. Ling, "On the theory and construction of k-clusters," *The Computer Journal*, vol. 15, no. 4, pp. 326–332, 01 1972. [Online]. Available: https://doi.org/10.1093/comjnl/15.4.326

[8] R. Almaghairbe and M. Roper, "Automatically classifying test results by semi-supervised learning," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 116–126.

[9] ——, "Separating passing and failing test executions by clustering anomalies," *Software Quality Journal*, 2017.

[10] M. Roper, "Using machine learning to classify test outcomes," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2019, pp. 99–100.

[11] C. Geethal, "Training automated test oracles to identify semantic bugs," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1051–1055.

[12] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, and M. Souza, "A machine learning approach to generate test oracles," ser. SBES '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 142–151. [Online]. Available: https://doi.org/10.1145/3266237.3266273

[13] A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, and M. Arratibel, "Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 30–39.

[14] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs." New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[15] M. Martinez, T. Durieux, J. Xuan, R. Sommerard, and M. Monperrus, "Automatic repair of real bugs: An experience report on the defects4j dataset," 2015.

[16] M. Böhme, C. Geethal, and V. Pham, "Human-in-the-loop automatic program repair," *CoRR*, vol. abs/1912.07758, 2019. [Online]. Available: http://arxiv.org/abs/1912.07758

[17] M. R. Hoffmann, E. Mandrikov, and M. Friedenhagen, “Jacoco java code coverage library,” 2022. [Online]. Available: https://www.jacoco.org/jacoco/trunk/index.html

[18] A. Groce, T. Kulesza, C. Zhang, S. Shamasunder, M. Burnett, W.-K. Wong, S. Stumpf, S. Das, A. Shinsel, F. Bice, and K. McIntosh, “You are the only possible oracle: Effective test selection for end users of interactive machine learning systems,” *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 307–323, 2014.

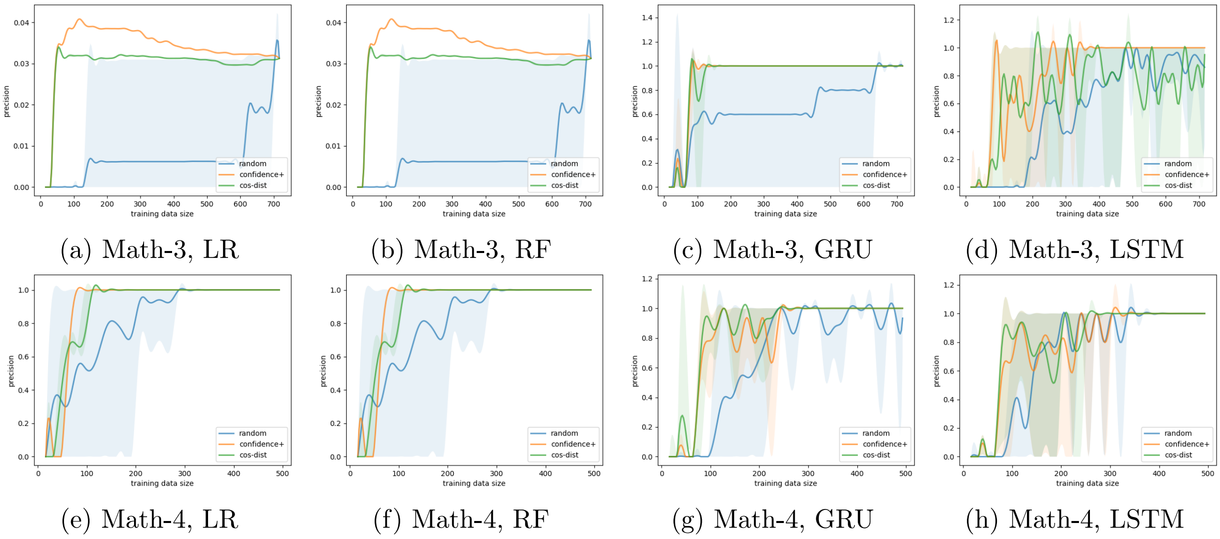
[19] Q. Wang, L. Gu, M. Xue, L. Xu, W. Niu, L. Dou, L. He, and T. Xie, “Facts: Automated black-box testing of fintech systems,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 839–844. [Online]. Available: https://doi.org/10.1145/3236024.3275533

[20] T. Jin, Q. Wang, L. Xu, C. Pan, L. Dou, H. Qian, L. He, and T. Xie, “Finexpert: Domain-specific test generation for fintech systems,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 853–862. [Online]. Available: https://doi.org/10.1145/3338906.3340441

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

A. Additional Results for Active Learning Experiments

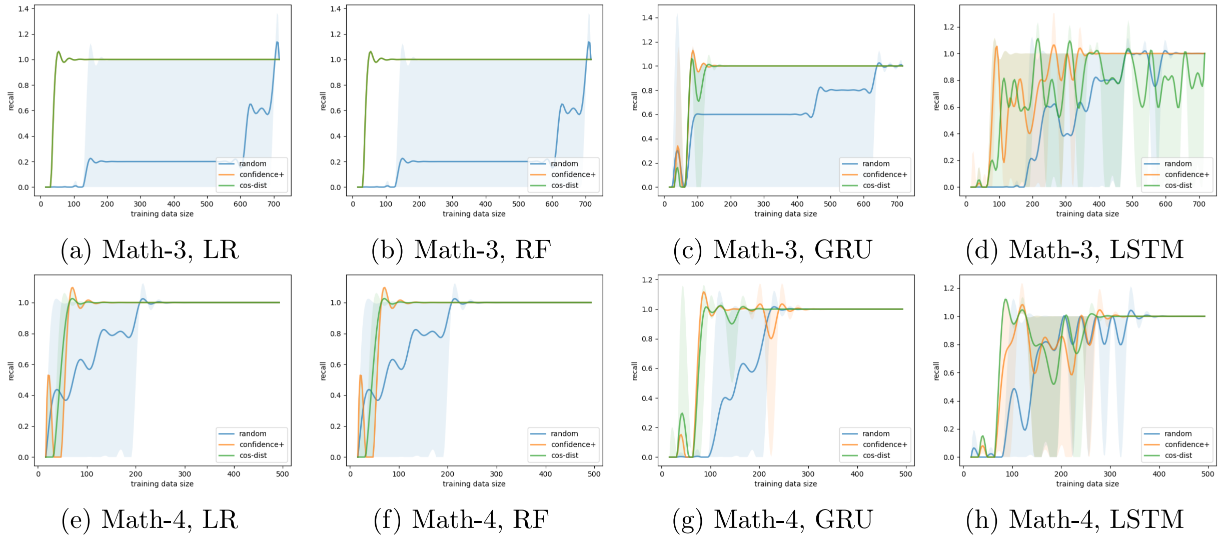
Due to lack of space, figure 3 only shows the F1-score of 2 of our SUT. Figure 4 and 5 shows the precision and recall curve with respect to each training set size. We can see that both confidence+ and cos-dist method outperforms random selection, and they tend to produce a stable result. Compare to cos-dist, confidence+ performs slightly better. As confidence+ is more time-efficient than cos-dist, it is the most preferred method in practice.



(a) Math-3, LR (b) Math-3, RF (c) Math-3, GRU (d) Math-3, LSTM

(e) Math-4, LR (f) Math-4, RF (g) Math-4, GRU (h) Math-4, LSTM

Figure 4: Precision curve for projects Math-3 and Math-4, with respect to each machine learning algorithm and active learning technique.



(a) Math-3, LR (b) Math-3, RF (c) Math-3, GRU (d) Math-3, LSTM

(e) Math-4, LR (f) Math-4, RF (g) Math-4, GRU (h) Math-4, LSTM

Figure 5: Recall curve for projects Math-3 and Math-4, with respect to each machine learning algorithm and active learning technique.