CS454 AI Based Software Engineering

Coursework #2: Learning to Rank

20150751 최상진

1. Introduction

Fault localization aims to support the debugging activities of human developers by highlighting the program elements that are suspected to be responsible for the observed failure.[1] In this coursework our goal was to implement a learning mechanism for automatic debugging using 33 existing SBFL scores and 8 code and change metrics for 156 real world faults from the Defects4J repository. Genetic programming is used for learning and training the model from the given dataset. After reviewing the paper given in the coursework document, I experimented with the methods that the paper used and added some variations to those methods to reduce overfitting and further optimize the ranking model.

2. SBFL scores & Code and Change Metrics

Spectrum Based Fault Localization (SBFL) is an existing localization technique that aims to assist debugging by applying risk evaluation formulae to program spectra and ranking statements according to the predicted risk.[2] It uses program spectra to predict the suspiciousness of each program statement to have a fault. In this coursework, 33 existing SBFL formulae are used for the training dataset.

Code and change metrics are features in the program that could give further indications of the suspiciousness of the program statement. There are 8 code and change metrics given as the training dataset which are related to age, churn, and complexity of the program element.

[1] J. Sohn and S. Yoo. FLUCCS: Using code and change metrics to improve fault localisation. In Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2017, pages 273-283, 2017.

[2] S. Yoo: Evolving Human Competitive Spectra-Based Fault Localisation Techniques. Search Based Software Engineering: Fourth International Symposium, SSBSE 2012, Riva del Garda, September 28-30, 2012, Proceedings, page 244

3. Learning with Genetic Programming

From 33 SBFL scores and 8 code and change metrics, I used genetic programming as the course document suggested. The GP algorithm evolves an expression which takes all the features for terminal values. I will start by explaining the ideas that I used from the paper, and further explain the the variation that I made to reduce overfitting.

I used the DEAP library, a Python evolutionary computational framework, to implement the genetic programming algorithm. I used a tree-based GP with single-point crossover with rate of 1.0 and subtree mutation with rank 1.0. The initial population is 40 and the population is initialized by the ramping method. The maximum tree depth is 8 and the GP algorithm stops at the 100th generation. The below table is the operators that I used for the GP algorithm. There is also an additional constant 1.0 as a terminal node.

Operator	Definition
Add	$a + b$
Sub	$a - b$
Mul	$a \times b$
Div	1 if $b = 0$, a/b otherwise
Neg	$-a$
Sqrt	$\sqrt{\lvert a \rvert}$

I will now explain the variations that I have made to reduce overfitting. Instead of calculating a single fitness by the average ranking of the faulty program element, I calculated two fitness values and designed the GP to be multi-objective. The first fitness function is same as the paper which is the average ranking of the faulty program element. The second fitness function is the standard deviation of the ranks of all the faulty program elements calculated by the GP expression. The reason for using the standard deviation of the rank of the faulty element is to make sure that the all the faulty elements in the program get some chance to get minimized which could overall reduce overfitting.

After experimenting the single objective GP that has a fitness function for average

rank, I realized that the rank of some faulty elements are easily minimized compared to other faulty elements. Thus, as the rank of those faulty elements get closer to 0, some outlier elements stay at a much higher rank. Since the fitness is the average of all the ranks of the faulty methods, after the ranks of faulty elements that are easily minimized go to 0, the outlier faulty elements don't have a chance to get minimized. Those outlier faulty methods is easily neglected by the GP algorithm.

When having a standard deviation of the as an additional fitness, the GP algorithm tends to minimize the rank of outlier faulty elements by reducing the standard deviation of the ranks of all faulty elements. The ranks of the easily minimized faulty elements are minimized slower and avoids overfitting compared to that performed by single objective GP.

I gave different weights to the two fitness: 5 for the average rank and 1 for the standard deviation. If the weight of the two fitness values are same or the weight of the standard deviation fitness value is higher, the GP algorithm could focus to much on minimizing the standard deviation and not much on the initial goal which is to minimize the ranks of the faulty methods. I have experimented with 5 variations of weights for the fitness functions: 10 to 1, 5 to 1, 3 to 1, 1 to 1, and 1 to 2. Five for the average rank and one for the standard deviation performed best in reducing overfitting and minimizing the ranks of the faulty methods.

I did not sampled the faults for the training dataset for fitness evaluation in each generation in GP. Instead I used all the faults in the training dataset for fitness evaluation. The paper sampled 30 faults from the total training dataset. But there are only 156 faults given as the training set, even more smaller than the training dataset used by the paper. After experimenting by sampling faults, I thought that because the dataset was so small, getting good performance with withheld faults would be too hard when sampling faults from the dataset. It could reduce overfitting, but I thought that achieving the original goal was more important. When there are more than 1000 faults, I will consider sampling the faults in the training dataset.

Instead of using elitism, preserving the best individuals from the parent generation, I used tournament selection of size 8. The tournament selection algorithm selects the best

individual among tournament size of 8 randomly chosen individuals out of 40. The reason is similar for using standard deviation for an additional fitness function. The ranks of the easily minimized faulty elements are minimized slower and the overall model avoids overfitting.

4. Evaluating Metric

I used accuracy(acc@n) in the validation step to analyze the performance of GP which is one of the evaluation metrics introduced in the paper. Acc@n counts the number of faults that have been localized within top n places of the ranking. The important thing to note is that when there are multiple faults, we assume that the fault is localized if any of them are ranked within top n places. The evaluation metric is slightly different in ranking the multiple faults with the training process. In the training process, if there are multiple faults, the fault with the highest rank is considered.

5. Validation

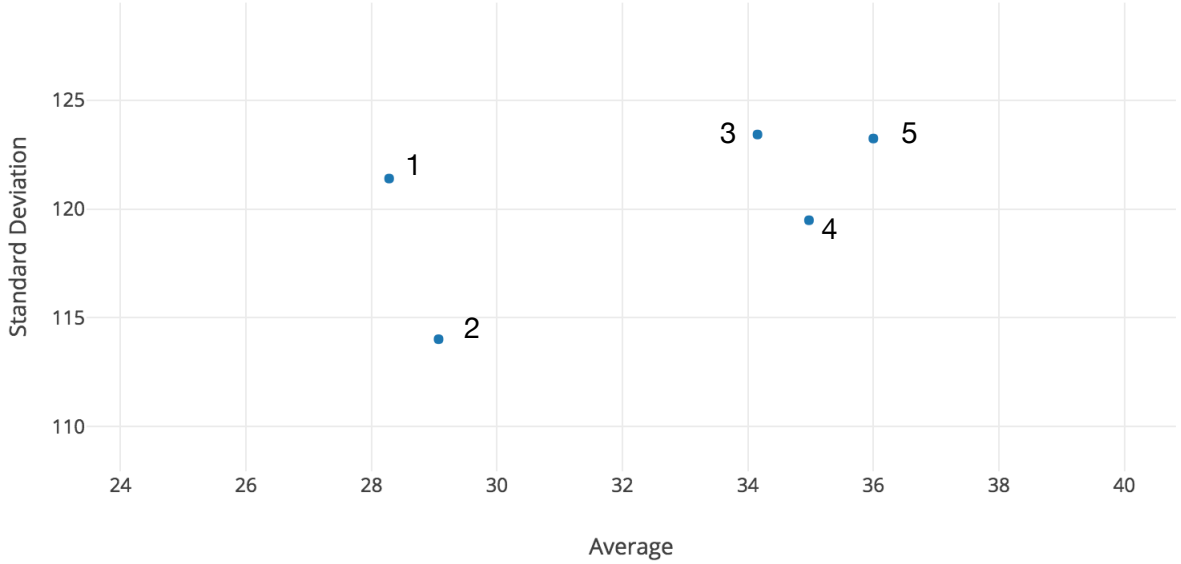
To validate that the GP algorithm performs well with withheld faults, I used five-fold cross validation. I randomly sampled 30 faults from the set of 156 faults. From the 30 faults, I divided the fault dataset into 5 different sets, each consisting 6 faults. In each run, 4 sets are used for training and 1 withheld set is used for validating and evaluating the accuracy metric. I executed 5 runs with the GP algorithm each for 30 generations and added all the accuracy evaluating metrics for 5 runs. Below table is the accuracy metric value for the validation experiment showing that the GP algorithm performs well with withheld faults.

		acc@1	acc@3	acc@5
1st run	**6 faults**	3	6	6
2nd run	**6 faults**	3	4	4
3rd run	**6 faults**	2	4	4
4th run	**6 faults**	6	6	6
5th run	**6 faults**	1	3	6
Total	**30 faults**	15	23	26

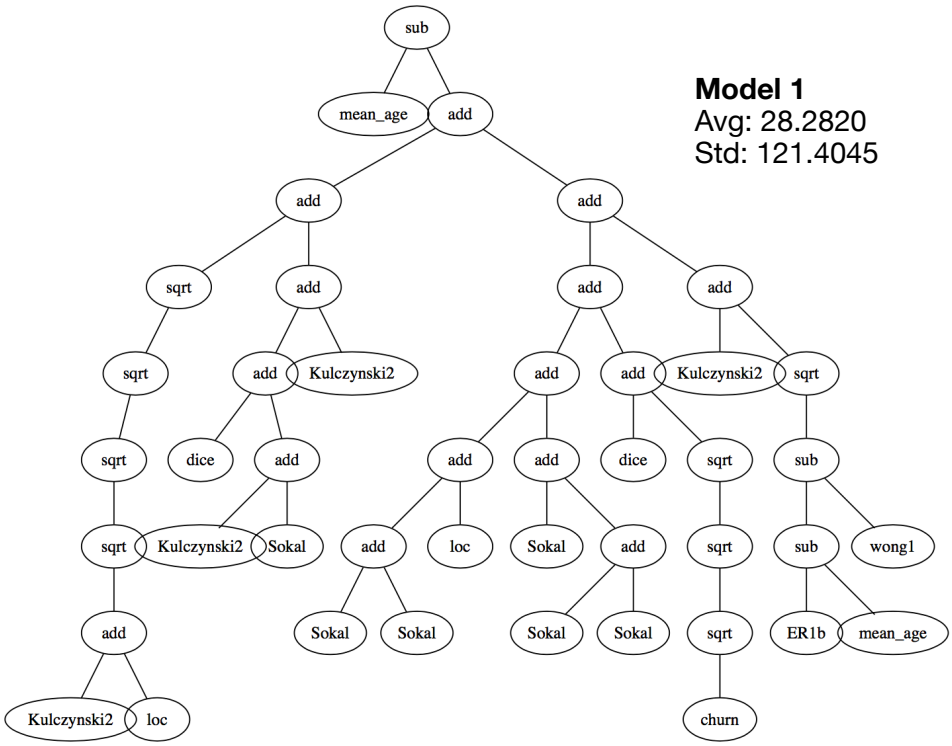
6. Results

Below is the five best trained model based on the two fitness values. There is no model that dominates the rest of the other models. Model 1 and 2 is the Pareto front. Thus, the final model that I should consider as the best trained model are model 1 and 2.

	1	2	3	4	5
Average	28.2820	29.0705	34.1538	34.9743	36.0
Standard Deviation	121.4045	114.0156	123.4294	119.4863	123.2457

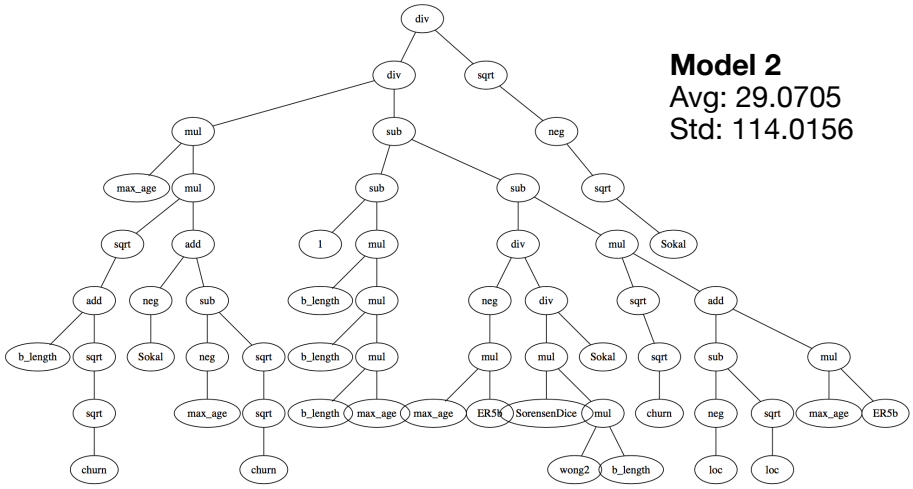


Between selecting from the two best trained models, I considered the diversity of the terminals in the expressions. I thought that the more diverse the terminals are, the better performance for withheld faults. The diversity of the terminals would reduce overfitting. If one of the SBFL scores or code and change metrics appear frequently, it could be an indicator for overfitting. Below is the graph of model 1 and 2.



Model 1
Avg: 28.2820
Std: 121.4045

	Sokal	Kulczynski2	dice	mean_age	loc	ER1b	Wong1	Churn
Model 1	6	4	2	2	2	2	1	1



Model 2
Avg: 29.0705
Std: 114.0156

	max_age	b_length	sokal	churn	loc	ER5b	SorensenDice	Wong2
Model 2	5	5	3	3	2	2	1	1

The diversities of the terminals were similar in the two models. For model 2, however, the 2 most used terminals were max_age and b_length. Out of 22 terminals, 15 of them are code and change metrics. I thought that using the code and change metrics highly can be an indicator for overfitting. For model 1, 5 out of 20 are code and change metrics. Therefore, I choose model 1 to be the best trained model.

Below is the accuracy(acc@n) evaluation metric for model 1 and 2. I did not considered the accuracy evaluation metric to select the best trained model for withheld faults between model 1 and 2, because I used all 156 faults in training the two models.

	acc@1	acc@3	acc@5
Model 1	90	126	139
Model 2	84	117	135

7. Testing

I performed a python unit test for testing the codes. I only tested the parse method from the Dataset class. For the genetic programming functions, since I used the DEAP framework, I was not sure how to generate the inputs and outputs to test the function. There is no output value for the validation function, so I could not test the function. To test the code, “fluccs_data/Lang_1.csv" and “fluccs_data/Lang_20.csv" should be in the same directory with “test.py”.

8. Conclusion

I was skeptical at first about using code and change metrics as a guidance for predicting the faults of each program statement. And I am still suspicious now. I did not quite understand why the Fluccs paper considered cases for multiple faulty elements differently with acc@n evaluation metrics and the training process. In the acc@n metric, when there are multiple faulty elements, fault is localized if any of the faults are ranked within top n places. I strongly believe that this should be changed. If there are multiple

faulty elements, one with the highest rank should be considered because that's how we trained the model. Also, there is a grammatical mistake in 278 p. in the Fluccs paper. "Given the set of 210 faults, we divides fault data set into ten different sets … ." I think divides should be changed to divide. Overall, I enjoyed having a chance to use the DEAP framework to train the learning algorithm.

9. References

[1] J. Sohn and S. Yoo. FLUCCS: Using code and change metrics to improve fault localisation. In Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2017, pages 273-283, 2017.

[2] S. Yoo: Evolving Human Competitive Spectra-Based Fault Localisation Techniques. Search Based Software Engineering: Fourth International Symposium, SSBSE 2012, Riva del Garda, September 28-30, 2012, Proceedings, page 244