Automated Regression Test Suite Optimization based on Heuristics

Dhanyamraju S U M Prasad, Simy Chacko, Satya Sai Prakash Kanakadandi, Gopi Krishna Durbhaka
HCL Technologies, India
Email: prasad@hcl.com, simy_c@hcl.com, satyasaiprakash.k@hcl.com, gopikrishna.d@hcl.com

Abstract — In the Software Development Life Cycle, Testing is an integral and important phase. It is estimated that close to 45% of project cost is marked for testing. Defect removal efficiency is directly proportional to the rigor of the testing and number of test cycles. Given this prelude, important optimization dual is to reduce the testing time and cost without compromising on the quality and coverage. We revisit this popular research and industry sought problem, in the historical data perspective. Proposed model has two steps. N Test cases based on multiple heuristics are recommended as part of first step. These heuristics can be derived based on test manager, test lead and/or test director requirements as inputs. The N test cases that are to be recommended will be derived upon executing evolutionary randomized algorithms such as Random Forest / Genetic Algorithm. These algorithms fed with historically derived inputs such as test case execution frequency, test case failure pattern, change feature pattern and bug fixes & associations. The recommended test suite is further optimized based on a 2 dimensional approach during second step. Test case specific vertical constraints such as distribution of environments, distribution of features as well as test suite composition parameters such as golden test cases, sanity test cases, that serves as horizontal constraints.

Keywords — Test coverage; Regression Test Suite; Random Forest; Test failure prediction; Optimisation; Greedy Algorithm;

I. INTRODUCTION

There are different types of testing as part of Software Testing Life Cycle such as Unit Testing, Sanity Testing, Functional Testing, Integration Testing, Load Testing and Regression Testing. Regression testing is a maintenance activity that is performed to ensure the validity of modified software [11] before the release. The scope of this paper is confined to the Regression testing alone.

Many researchers have been working in this area and different techniques have evolved. Generally, coverage and redundancy of the test cases is analyzed and considered in regression test suite minimization. We proposed relevancy based metric that is derived from historical executions of the test cases.

Usually regression testing tends to be elaborate involving large number of test cases in any mid-level to large enterprise software products. Instead of executing all the test cases for regression, selecting a subset of test cases and plan them accordingly to execute on different environments so as to identify and cover all the fault scenarios based on the risk value of the test case is a gargantuan task. Instead of adopting most literature referred methods such as coverage based and fault counts, our method tries to automatically associate important test cases to file changes across builds. Based on these derived correlations system recommends most relevant test cases to the new regression build. Second level of optimization matches the requisite maximum number (N) that can be executed by the testing team before the hard release date and time. Our work also assumes importance in the context of the below claim from the reference [15].

“Out of the 159 papers surveyed, only 31 papers list a member of industry as an author or a coauthor. More importantly, only 12 papers consider industrial software artefacts as a subject of the associated empirical studies.” [15]

In this paper we shall discuss how the process of test suite minimization had been achieved for regression testing. Test recommendation and optimization helps the whole testing community to estimate both time and effort to execute the regression plan. Test suite minimization comprises of 2 modules enlisted below:

1. Analyzer Module
2. Optimizer Module

Analyzer Module comprises of evolutionary algorithms such as Random Forest / Genetic Algorithm. This has been detailed in the analyzer module section.

Optimizer Module comprises of greedy algorithm executed in a 2 dimensional approach. This has been detailed in the Optimizer module section. There were different datasets on which this technique had been applied to two cases.

Case-1: Open source dataset of which the results have been shared with in this paper. This data comprised of 8 Releases and 15 builds with 207,881 source files along with 2,942 test cases.

Case-2: In the other case, where multiple enterprise datasets had been considered. One of the dataset metrics is being shared herewith where in minimum of 18 Releases, 278 Builds, 10,117 File Changes, 25,330 file
changes across builds, 994 test cases, 3,899 executions have been considered.

II. LITERATURE REVIEW

A brief note on the algorithms has been enlisted below, along with few cases discussed where they had been applied in the past to resolve similar issues such for Regression Test Suite Minimization.

**Random forests (RF)** [5] are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. The generalization error for forests converges to a limit as the number of trees in the forest becomes large. The generalization error of a forest of tree classifiers depends on the strength of the individual trees in the forest and the correlation between them.

Random Forests [6] are a type of ensemble method which makes predictions by averaging over the predictions of several independent base models. Since its introduction by Brieman (2001) the random forests framework has been extremely successful as a general purpose classification and regression method.

**Genetic algorithm** (GA) is a population-based search method. Genetic algorithm requires several parameters including the following:

1. Maximum number of generations,
2. Population size,
3. Crossover rate,
4. Mutation rate,
5. Convergence criterion

The GA maintains a population of n chromosomes with associated fitness values [8]. Parents are selected to mate, on the basis of their fitness, producing offspring via a reproductive plan. Consequently highly fit solutions are given more opportunities to reproduce, so that offspring inherit characteristics from each parent. As parents mate and produce offspring, room must be made for the new arrivals since the population is kept at a static size.

Individuals in the population die and are replaced by the new solutions, eventually creating a new generation once all mating opportunities in the old population have been exhausted; In this way it is hoped that over successive generations better solutions will thrive while the least fit solutions die out.

New generations of solutions are produced containing, on average, better genes than a typical solution in a previous generation. Each successive generation will contain more good "partial solutions" than previous generations. Eventually, once the population has converged and is not producing offspring noticeably different from those in previous generations, the algorithm itself is said to have converged to a set of solutions to the problem at hand.

III. HISTORICAL BUILDS DATA ANALYSIS

The analyzer module comprises of a model (i.e. an algorithm) that shall be executed to compute the execution frequency of the test cases and derive the high executed test cases based on past history and also a threshold value stated by the Test Manager.

The inputs to the model comprises of the Source code File Changes (FCs) and executed Test Cases (TCs) so as to learn from the past history (i.e. learn builds) which shall be considered for learning.

The execution of a model is classified into two approaches based on the input that shall be provided to it. They are with source and without source. Sometimes source code might not be available or protected and hence only the execution history of Test cases executed across the builds may be available. To handle such scenarios another approach known as without source shall be followed here.

When the source code as well test cases are available then few of the analytics models such as RF, GA, Hopfield, SVD, and FP Growth shall be executed. The decision of the model selection shall be done based on the goodness of fit test (i.e. the model that provides higher learning cum prediction accuracy). In this paper we restrict the scope of discussion to the models RF and GA alone.

On execution of the model the test cases recommended along with the execution frequency computed by the analyzer for the test cases shall be considered here as an additional weightage for re-enforcement during Optimization.

When the source code is not available and only TCs are available then the statistical model shall be executed. The execution frequency of the TCs shall be considered here as an additional weightage for re-enforcement during Optimization.

IV. ALGORITHM EXCERPTS

The Execution approach of the different models applied here has been discussed.

Random forest is applied to the source code File changes and the executed test cases so as to complete the learning and get the random formulae generated. This had been used to predict the test cases of the new build by giving its source code file changes.

**Pseudo Code of RF Model** is shared below:-

**Inputs**
File changes of learning builds [FC],
Test cases of learning build [TC],
New File changes of Test build [New FC],

**Execution**
1. Generate the binary of FC and TC
2. Get the unique and duplicate patterns of TC
3. Repeat for t unique patterns of TC binary
   a. Consider each TC(t) pattern & FC.
b. Generate the Random formulae and predict the TCs
   \[ \text{data} = \text{TC}(t) \cup \text{FC} \]
   \[ RF = \text{random Forest} \left( \text{TC}(t) \sim., \text{data, NTree}=101 \right) \]
   \[ \text{Predict Result} = \text{predict} \left( RF, \text{transpose} \left( \text{New FC} \right) \right) \]

4. Execute RF model based on TC(t) pattern
5. Apply the same value to the duplicate patterns based on each unique TC(t)

Initially, with the above approach every time learning and predicting together was consuming more time. As the number of file changes and test cases increased the time to learn and predict increase proportionately to 1 hour for RF and 6-8 hours for GA. Hence, to reduce time and reuse the model learn and test approach had been introduced.

Here, the in the learning phase generated results i.e. the random formulae or the co-efficient are stored within a signature module. During the first execution, the signature module generates 16-bit MD5 for the coefficients of the RF model.

Later, the respective signature models generated are used for the prediction of the test cases for the test build. This helps in reduction of time. Every time the combination of builds used for builds changes a new signature module is generated to store the model.

**Pseudo Code of RF Model with learn & Test:-**

1. Learn phase
   \[ \text{data} = \text{TC}(t) \cup \text{FC} \]
   \[ RF = \text{random Forest} \left( \text{TC}(t) \sim., \text{data, NTree}=101 \right) \]
2. Store the random formulae with the 16-bit MD5 based on the pattern using the signature
3. Complete the Learning phase
4. When predict is initiated the learn results are acquired comparing the 16-bit MD5 signature stored, and applied.
   \[ \text{Predict Result} = \text{predict} \left( RF, \text{transpose} \left( \text{New FC} \right) \right) \]

This model can be used to predict the test cases for the new source code file changes. To achieve better results with Random Forest Model, RF tuning was also done.

The model Random Forest is tuned further to improve the accuracy. RF tuning is done with the data set with different number of trees such as (101,201,301,401,501,601,701,801,901 & 1001) and respective mtry values with least “OOBError” for every iteration.

**Pseudo Code of RF tuning** is shared below:-

\[ \text{best}_mtryVal = \text{min}(mtry\_val) \]
\[ \text{for} \ iTree \ \text{to} \ \text{NTrees} \ [101:1001 \ \text{step} \ 100] \]
\[ \ \text{tuneRFMat} = \text{tuneRF fn} \]
\[ \ mtry\_val = mtry\_val \ \text{with} \ \text{min} \ "\text{OOBError}" \]
\[ \ \text{end for} \]
\[ \text{best}_mtryVal = \text{min}(mtry\_val) \]
\[ freq = \text{identify the frequency count of tree widely got tuned above} \]

Once the best fit Ntree and mtry value is derived for the data set this can be saved. The same can be used to for the RF model while learning. For prediction, only the co-efficient stored with the signature module are required. The error convergence is based on the OOB error. More no of trees are required to stabilize the error but not so many that would over correlate the ensemble, which leads to over fit.

The learning results of the above models have been saved with a signature module, which is later used for prediction of test cases.

The evaluated result of the predicted test cases using the RF model has been shown in the AUC plot in the Figure 1.

![Figure 1: AUC plot of RF Model](image)

Similarly, the pseudo code of Genetic Algorithm model applied for the test suite minimization has also been discussed below.

**Genetic Algorithm Pseudo Code**

1. Randomly initialize population(t) to the size of FC[train]
2. Determine fitness of population TC(t) sample pattern
3. Repeat for MaxIterations
   a. Select parents from population(t)
   b. Perform crossover on parents creating population (t+1)
   c. Perform mutation of population (t+1)
   d. Determine fitness of population (t+1) until best individual is good enough.
4. Converged test sequence coefficient is noted

The above pseudo code result completes the learning phase and produces the signature module with respective coefficients. Later, the respective signature models generated are used for the prediction of the test cases for the specified test build.
The evaluated result of the predicted test cases using the GA model has been shown in the AUC plot in the Figure 2.

V. RESULTS AND CASE STUDIES

The comparative metrics of learning error rate using RMSE of the evaluated RF and GA models have been enlisted build wise in the Table I. Five builds (B15-B20) have been considered here as a data set. The Build 20 has all distinct test cases which have not appeared in the past history.

<table>
<thead>
<tr>
<th>Builds</th>
<th>FC(Nos.)</th>
<th>TC(Nos.)</th>
<th>RF</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>B15</td>
<td>22229</td>
<td>670</td>
<td>5.3</td>
<td>0.1</td>
</tr>
<tr>
<td>B16</td>
<td>21757</td>
<td>670</td>
<td>9.8</td>
<td>6.7</td>
</tr>
<tr>
<td>B17</td>
<td>23570</td>
<td>513</td>
<td>6.6</td>
<td>11.2</td>
</tr>
<tr>
<td>B18</td>
<td>64941</td>
<td>602</td>
<td>6.8</td>
<td>9.9</td>
</tr>
<tr>
<td>B19</td>
<td>23165</td>
<td>411</td>
<td>5.4</td>
<td>7.3</td>
</tr>
<tr>
<td>B20</td>
<td>25016</td>
<td>76</td>
<td>5.5</td>
<td>5.9</td>
</tr>
</tbody>
</table>

The comparative metrics of learning accuracy of the RF and GA models have been enlisted consecutive build wise in Table 2. The Average is calculated with and without B20.

<table>
<thead>
<tr>
<th>Builds</th>
<th>FC(Nos.)</th>
<th>TC(Nos.)</th>
<th>RF</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>B15</td>
<td>22229</td>
<td>670</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>B16</td>
<td>21757</td>
<td>670</td>
<td>70%</td>
<td>80%</td>
</tr>
<tr>
<td>B17</td>
<td>23570</td>
<td>513</td>
<td>100%</td>
<td>24%</td>
</tr>
<tr>
<td>B18</td>
<td>64941</td>
<td>602</td>
<td>91%</td>
<td>67%</td>
</tr>
<tr>
<td>B19</td>
<td>23165</td>
<td>411</td>
<td>100%</td>
<td>98%</td>
</tr>
<tr>
<td>B20</td>
<td>25016</td>
<td>76</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>77%</td>
<td>61%</td>
</tr>
<tr>
<td>Average without B20</td>
<td></td>
<td></td>
<td>92%</td>
<td>74%</td>
</tr>
</tbody>
</table>

The Figure 3 shows the comparative plot holding the average that is calculated without B20.

VI. TEST SUITE OPTIMIZATION

The recommended test suite is further optimized based on a 2 dimensional approach. Test case specific vertical constraints such as distribution of environments, distribution of features as well as Test suite composition parameters, that serves as horizontal parameters.

This module shall help the Test Manager to plan the number of Test cases (N) to be executed in this regression plan.

This indeed also helps to plan the number of resources required and estimate the time to compete the task based on the execution time of a test case and many other parameters.

Moreover, the most relevant test cases are identified by computing the execution frequency, along with their derived risk status value. All the test cases executed on different environments along with their computed execution frequency, derived risk status value and their respective features shall be considered here as an input in addition to the Analyzer Module output to perform Optimization.

There are different types of test cases that shall be considered here and suggested by the optimizer module for execution based on the Test Manager distribution provided for each test case type among the horizontals. They are

- Golden Test Cases,
- Change based Test Cases,
- Sanity Test Cases,
- Environment Test Cases,
- Feature Test Cases,
- Bug Test Cases,
- Unstable Test Cases
A brief note on different test case types considered in the horizontals as stated above has been provided below:

The most frequently executed test cases repeatedly across all the Regression builds, which are necessary and should be executed, are known as Golden Test Cases.

The test cases executed for a particular class / method in the source code are derived by using an instrumentation approach are called Change Based Test cases.

The minimum essential test cases that need to be executed to evaluate the essential functionality are known as Sanity Test Cases.

The test cases that have been executed on the different environments are known as Environment Test Cases.

The test cases that are classified based on the different features mapped to them are known as Feature Test Cases.

The test cases derived based on bug / defect identified and later fixed in the new build or release in the past history shall be considered as a sudo test case. Hence these test cases are known as Bug Test Cases.

The failure pattern of test cases also known as Risk Status is derived from the execution history of test cases. This is to calculate the weightage for each test case and derive the most unstable test cases across the builds. Additionally, an affiliation score shall also be provided by deriving the trending pattern of the environments on which the test cases have been executed recently across the builds. These types of test cases are known as Unstable Test Cases.

There are multiple constraints in vertical that have been considered here for Optimization module as listed below:

A. Basically the number of Test cases (N) the Test Manager would like to execute as part of the regression plan.
B. Distribution percentage of Environments or Architecture over which the test cases need to be executed
C. Distribution percentage of Test case types that need to be executed
D. Affiliation score to the test case type also need to considered here
E. Any specific Feature, may be existing one that has been updated or a New Feature that has been introduced, which the Test Manager would like to execute for this release
F. When a new feature is introduced this should be compatible with all the environments. Hence in such cases the Test Manager would recommend the test cases associate with the new feature to be executed across all the environments based on the environment distribution suggested above. Hence, option for this is also provided and known as Repeat.
G. Minimum threshold of the computed execution frequency of the test cases
H. Failure rate i.e. more importance shall be given to test cases with higher risk rate first to be considered first or not.

The above constraints shall be applied to the input test cases of the optimizer module. Going with the greedy approach algorithm here every constraint need to be satisfied and shall be verified upon at each stage comparing it with minimum distribution (both environment distribution and test case type distribution) stated by the Test Manager. At any point, the minimum of test cases that need to be recommended should be greater than or equal to the optimization number N stated. Hence, constraint satisfaction and relaxation shall be applied by the optimizer module to meet the requirement of the constraints stated above. To balance the horizontal parameters here (considered as horizontal constraints i.e. Test Case Types) the minimum number of test cases that need to filled in for each type shall be distributed across, with the test cases that have satisfied the vertical constraints. The remaining test cases available in the vertical shall be distributed across the horizontal maintaining the cut-off specified as maximum number for each test case type. Additionally, scope to modify/update the suggested test cases based on the Test Manager’s past experience is also provided. On completion, the Regression Test plan suite shall be generated as a report. In parallel, the inputs at each stage both for analyzer module and optimizer module, suggested outputs of the models and also the respective changes done by the Test Manager to generate the Regression Test plan suite shall be stored so as to reproduce next time while generating another New Regression Test plan suite. One way this appears to be just reproducing that has been saved. But looking into the analytical perspective every observation is an additional input that can be used for learning (i.e. by giving additional weightage to specific test case which had been
additionally accepted/recommended by Test Manager) for the Model during the prediction and recommendation of the Test cases for the next time.

VII. CONCLUSION AND FUTURE WORK

The approach of test suite minimization along with the results of few models RF and GA for analysis and greedy approach for optimization have been discussed and published herewith. This approach has helped some of our clients to plan their regression testing well and improve their performance up to 12%.

We are trying multitude of algorithms including Single Value Decomposition, FP Growth. We are also evaluating the mutual dependencies, correlations and cohesion among various parameters such as frequency of execution, failure status and frequency, features and environmental dependencies.

ACKNOWLEDGEMENTS

Authors would like to thank Dakshin, Vijay Anand, Sandeep and Sukamal for their encouragement and critical inputs at various stages of development. We value and appreciate the work done by Srikanth, Sekhar for bringing out timely reports. We also thankfully acknowledge our customers who have adopted this solution and gave sufficient feedback.

REFERENCES


