Contents

1 Introduction

1.1 Domain Background

The topic for this capstone project is based in Software Testing. Specifically Spectrum-Based Fault Localization for larger projects. The required terminology is given by the following definitions.

Definition 1. *"Failures. A failure occurs when the user perceives that the program ceases to deliver the expected service." [1]*

Definition 2. *"Errors. A discrepancy between a computed, observed, or measured value or condition and the true, specified or theoretically correct value or condition. Errors occur when some part of the computer software produces an undesired state. Examples include exceptional conditions raised by the activation of existing software faults, and incorrect computer status due to an unexpected external interference. This term is especially useful in fault-tolerant computing to describe an intermediate stage in between faults and failures." [1]*

Definition 3. *"Faults. A fault is uncovered when either a failure of the program occurs, or an internal error (e.g., an incorrect state) is detected within the program. The cause of the failure or the internal error is said to be a fault. It is also referred as a "bug." " [1]*

1.2 Problem and Solution Statement

For a given program code containing at least one fault, the deployed Machine Learning algorithm should help localizing the faults in the code. In particular, the algorithm returns a priority list for a subset of all methods. A higher priority means that the method is more likely to contain the fault. Based on this priority list, multiple evaluation metrics can be applied. In order to localize the faults of a test case, the method spectrum as described in section 1.3 is used. More specifically, the localization is trained and tested on the Defects4j dataset. The proposed algorithm should be able to localize the faults for the different test cases better than the baseline benchmark.

1.3 Data

The Data is given by the method coverage data of the Defects4j dataset. The Defects4j dataset is a collection of natural faults of six Open Source java projects. These being Char, Lang, Time, Closure and Mockito. Every fault is contained by itself and contains no other

	m_1	m_2	m_3	p
u_1	0	1	0	-
u_2	1	0	1	+
u_3	1	0	1	+
u_4	0	1	1	-

Table 1-1: A spectrum data table containing the methods, the tests that executed them and the fail or pass value of the tests. Unit test one and three fail here. Here it would be likely that m_2 is faulty. [2]

faults. However a fault can be based on multiple methods or even classes. The fault is essentially given by the changes that were required in order to fix a given problem. Any method that required a change is considered faulty. [2]

The method coverage or method spectrum has variations in its definitions based on the literature and program. For this project, method spectrum is defined to align with the output of GZoltar, a program to generate coverage data from the code of a java project. In the following, the method spectrum is a collection of tuples. Every tuple describes a unit test of the given project. Thus, the method spectrum contains u tuples where u is the number of unit tests. A tuple contains $m + 1$ values where m is the number of methods in a project. The first m values are boolean values and describe whether method m_i has been executed by the unit test u_j. The last additional value p reflects whether a test has failed or passed and is given by $-$ and $+$ respectively. An example can be seen in Table 1.3. Please note that the method spectrum will also be referred to as *spectrum data* in this proposal. This definition is closely related to the definition found in [2].

In order to restrict the amount of data required for training and testing, only a subset of the Defects4j Dataset is used. Otherwise the dataset would heavily exceed the soft limitations of the capstone project. The training data chosen is a mix between hand picked and random faults of the Defects4J dataset. The testing data on the other hand is the collection of all Mockito faults excluding fault 5 and 26. This is due to the nature of the bug not showing up in the coverage data at all. Thus, those faults are not localizable by coverage based fault localization. Table 1.3 shows which faults are chosen from which Defects4j project. Please note that no validation data is used here, since every test case is self contained and relearnt from scratch. This is further explained in 3. The following website is a useful resource in order to gain insight into the faults of Defects4j (https://github.com/Spirals-Team/defects4j-presentation-urls).

Additionally, some manually adjusted files containing the lists of faulty elements for every

Project	Faults
Chart	$2, 14, 21, 25$
Time	$1, 3, 5, 9, 18, 21, 23$
Lang	$5, 23, 30, 32, 36, 41, 63$
Math	$6, 8, 16, 38, 51, 64, 66, 71, 92, 97$
Closure	$3, 16, 26, 27, 37, 43, 47, 49, 72, 76, 103, 108$
Mockito	$[1 - 38]/\{5, 26\}$

Table 1-2: The list of faults chosen from the defects4j dataset.

chosen test case are used. These are simply used to make the evaluation easier and reduce the data, as they could be read from the Defects4j dataset, which is too large for the purpose of this project. This data will be referred to as *fault data*.

2.1 Evaluation Metrics

In order to evaluate the output, two metrics are deployed. The first metric is the so called *first hit*. Given a priority queue of methods and their likeliness to be faulty, the first hit is the position of the first faulty element. The collection of faulty methods is given by the fault data and the full list is given by the spectrum data. The priority queue is required to consist as a subset of these methods. For example if element 5, 43 and 321 of the priority list are actually faulty, then the first hit result is 5. From intuition this is done because finding a fault typically leads the programmer to find other methods related to that fault.

The second metric is *wasted effort*. Wasted effort is the number of correct lines that are contained in all elements before the first hit. The intuition here is that the fault localization is supposed to be used in debugging. Thus, having a high wasted effort is equal to wasted time for some employee debugging the application. Taking the same example from the paragraph above, the wasted effort would be the number of lines of code in the first four methods of the priority queue.

As a final evaluation, the average over multiple test cases can be used to give an overall effectiveness of the proposed algorithm. This is only an extension of the previous two metrics, though.

2.2 Benchmark Model

As models of comparison Tarantula and Ochiai are used. The result is a similar priority queue of elements suspected to be faulty. They are the most common baselines for fault localization and are well defined on any model and can be reproduced easily. The mathematical formulation of Tarantula and Ochiai are given by Equation 2-1 and 2-2.[2] Both metrics are executed for every method (column) in the spectrum data. Variable c_{ef} is the overall number of executions (e) by failing (f) test. Equivalently, c_{ep} is the overall number of executions by passing (p) tests and c_{np}/c_{nf} similarily are the overall numbers of passing or failing tests that did not (n) execute the method. Evaluating this for a column returns a suspiciousness value for a method, which can then be inserted in a priority queue.

$$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ep}}{c_{ep}+c_{np}}} \tag{2-1}$$

$$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf})(c_{ep}+c_{np})}} \tag{2-2}$$

3 Project Design

3.1 General Approach

When looking for a general approach, I had a look at multiple papers including approaches using *feature selection*, *neural networks*, *clustering* and *genetic algorithm*. Finally, the neural network approach given in [3] was chosen. It is an interesting and promising approach which has not been explored too much. Additionally, since the paper is a few years old, its usage of neural network is outdated. This means there is possibility for improvement. Furthermore, it was evaluated on the Siemens and Space dataset, which allows me to add something by evaluating an extended approach on the more modern Defects4j dataset.

The approach given by [3] uses neural networks in conjunction with a *virtual test set*. This virtual test set is an extension of the original spectrum data. The main difference is that the virtual test set only contains unit tests that evaluate only one method. Of course no such test exists, but the neural network is supposed to attach a suspiciousness value to every element of the virtual test set. The suspiciousness value describes how likely the algorithm predicts a method to be faulty. Since every element there corresponds to only one method, the priority queue can be built using the prediction of the neural network. Doing it like this, a typically unsupervised learning problem has been turned into a supervised learning problem. An example of the virtual test set can be seen in Table 3.1

3.2 Outlining the Algorithm

3.2.1 Implementation

Given the method spectrum data and the virtual test set, a priority queue should be retrieved. In order to reduce the number of components, only the union of components that were executed in some faulty test are considered. A method can not be faulty by our metrics if no faulty test case executes it. Afterwards, clustering is applied to the tests (lines) of the

	m_1	m_2	m_3	m_4
vt_1	1	0	0	0
vt_2	0	1	0	0
vt_3	0	0	1	0
vt_4	0	0	0	1

Table 3-1: A spectrum data table containing virtual tests that only execute one method. Their fail or pass is unknown and is supposed to be predicted by the neural network.[3]

data using the EM-Algorithm. My idea of how to apply the clustering to the data is new to the virtual test set appraoch. In particular tests currently only fail or pass ($-$, $+$). However, based on the distance to faulty methods this value shoud be adjusted and instead be given by a float instead of a boolean. While this is not sure to work, this will be thouroughly tested. In case this does not work, other clustering techniques will be tested. Please note that the proposed approach could end up being more similar to K-NN than to clustering depending on how the augmentation of the data is done.

As another additional preprocessing step, PCA without reducing the number of components is considered. ICA or RCA will definitely not work as they will destroy the idea of the virtual test set. PCA only rotates the data and aligns it in the direction of maximal variance. This should result in the virtual test set being more expressive than previously. While this may destroy the possibility to localize some virtual tests, it should help with the localization of others.

As the final and primary step, the neural network is deployed. The neural network learns to predict the modified fail or pass value ($[0, 1]$). Due to the way this behaves, several things that are normally included in a neural network are going to miss. For example, there is no typical overfitting. The model is retrained for every test case and is only based on that spectrum data. So overfitting is only defined in respect to the virtual test set for every case and not as a global metric for evaluation. This overfitting can vary from fault to fault and needs to be adjusted for by some parameters. The plan currently is to dynamically adjust the epoch number during run time based on data size, current epoch number and maybe some other parameters. The neural network itself is planned to consist of 3 fully connected layers with batch normalization, dropout and leaky ReLU activation. The sizes will be automatically adjusted based on the size of the method spectrum for a given test case. The loss function will most likely end up being a weighted cross entropy loss. The reason is the importance of recall in the given problem. Faulty elements are important and need to be found, the precision here can nearly be disregarded. Thus, the weighted cross entropy loss probably uses a strong recall weight.

Depending on the amount of time left, additional feature selection as a preprocessing step will be taken into consideration. However, that will most likely be outside the scope of this project.

3.2.2 Analysis and Visualization

While the clustering step is a preprocessing step, it will most likely be implemented and evaluated last. First, a baseline implementation of the neural network is used to do the first fault localizations. When the neural network is capable of localizing faults, the clustering algorithm and the PCA step is added. The algorithm then is evaluated using different combinations of hte proposed algorithms, in- and excluding PCA and different clustering/K-NN techniques. This results in multiple valid outputs, that can be compared with each other.

In terms of visualization, the ranking of faulty methods in the priority queue over time can be tracked, resulting in line graphs showing the way faulty methods move inside the priority queue. Additionally, almost all outputs are in the form of tables. This includes for example the first hit and the wasted effort metric, where this can be done for every fault of defects4j. The priority queue could also be visualized as a heat map, although that is not necessarily sensible.

Bibliography

[1] Lyu, M. R., ed.: Handbook of Software Reliability Engineering. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[2] Souza, H. A. de; Chaim, M. L.; Kon, F.: Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. In: CoRR vol. abs/1607.04347 (2016). arXiv: 1607.04347. http://arxiv.org/abs/1607.04347.

[3] Zheng, W.; Hu, D.; Wang, J.: Fault Localization Analysis Based on Deep Neural Network. In: Mathematical Problems in Engineering vol. 10.1155/2016/1820454 (2016). http://dx.doi.org/10.1155/2016/1820454.