Contents

1 Introduction

1.1 Project Overview

The project is based on Software Testing. In particular, the topic is Fault Localization. For a given faulty code, the general problem is finding the method or even specific lines that contains the fault. In order to build models for Fault Localization, some datasets were gathered the past few years. The most common dataset are Space, Siemens and Defects4j, which all are based on java. For the purpose of this project, Defects4j is used. In order to understand the given problem statement, some definitions are required. The definitions follow the standards for software testing.

Definition 1. *"Failures. A failure occurs when the user perceives that the program ceases to deliver the expected service." [3]*

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." [3]*

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." " [3]*

1.2 Problem 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. This is also called the suspiciousness of a method. 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 2.1 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 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.

For the first hit, the average ranking/wasted effor is used. This only has an effect in case of ties. In particular, the average between the best and the worst ranking is rounded down in order to retrieve the ranking of a particular component. For example a list of four components with a suspiciousness value of 0.5 each would create a tie. The best possible rank assigned to any of the components is one, since all of them tie with the first rank. The worst possible rank is four, because they also tie with the fourth component. The resulting average ranking for all of them is two, since the average between one and four is rounded down.

Originally, the proposal had the added metric of *wasted effort*. This has been replaced, since the wasted effort was mostly identical to the first hit except for some minor differences. Thus, it did not help get more insight into the model. The wasted effort was replaced by a metric to rate the model overall for multiple faults. In particular, they are called the *average first hit* and the *normalized average first hits*. For multiple faults, the average first hit is simply the average of all first hits for the training/testing set. The normalized average takes the number of components for a particular fault into account. Equation 1-1 shows the mathematical definition for the normalized average. F_i is the first hit value of fault i. Variable n_i is the number of component fault i contains.

$$norm_{avg} = \Sigma_{\forall i} avg(F_i/n_i) \tag{1-1}$$

2.1 Data Exploration and Exploratory Visualization

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 faults.[4] 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.

Figure 2-1: The Math 8 fault of the Defects4j dataset as found in [1]. The code lines indicated by color display the required changes to solve the fault.

An example for a fault in the dataset is given by Figure 2.1. The faulty lines are indicated by the their color. In this simple case, only one component is considered faulty. In particular, the *sample(int)* method is faulty in this case. This method was changed as indicated by the red and green lines. Green lines describe line additions whereas red lines describe line removals. All faults of the Defects4j dataset are similarily defined and can be seen in [1].

In order to represent the faults as useable data, the method coverage or method spectrumis used. The 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 2-1. The example does not correspond to any particular fault of the dataset, as any example would not be displayable due to size. Please note that the method spectrum will also be referred to as *spectrum data* in this project. This definition is closely related to the definition found in [4]. The number of components components can vary greatly between different cases. For example Math8 contains 22 components that are executed by some failing method, whereas Closure3 contains 1089 components.

	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 2-1: 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. [4]

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 12, 5 and 26. This is due to the nature of the bug (for 5, 12 and 26) not showing up in the coverage data at all. Thus, those faults are not localizable by coverage based fault localization. Table 2-2 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 Section 2.2.

Project	Faults
Chart	$2, 14, 21, 25$
Time	$1, 3, 5, 9, 18, 21, 23$
Lang	$5, 30, 32, 36, 41, 63$
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	$[1 - 38]/\{5, 12, 26\}$

Table 2-2: The list of faults chosen from the Defects4j dataset.

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

2.2 Algorithm Approach

2.2.1 General Approach

The approach given by [5] 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 2-3.

	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 2-3: 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.[5]

2.2.2 Outlining the Algorithm

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 data using Agglomerative Clustering. In particular tests currently only fail or pass ($-$, $+$). However, based on the number of failing test cases in the same cluster, this value shoud be adjusted and instead be given by a float instead of a boolean.

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. Thus, the parameters need to be adjusted dynamically and automatically for all faults.

The neural network itself consists of 3 fully connected layers with batch normalization, dropout and leaky ReLU activation. The sizes is automatically adjusted based on the size of the method spectrum for a given test case. The loss function used is 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.

2.3 Benchmark Model

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

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

For the benchmark models 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.[4] 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.

3.1 Data Preprocessing

The data is initially retrieved and stored as described in Section 2.1. The first optional preprocessing step applied to this is dimensionality reduction. This is done in a simple manner, based on the data spectrum. Any component that was not executed by a failing test case can not be faulty. Every line of data is adjusted such that only components executed by at least one failing test case are included in the data spectrum.

Afterwards, clustering is applied. This is done using Agglomerative Clustering in order to adjust labels. Labels previously were described by boolean values. Either pass or fail. Clustering is used to adapt the label of all passing test cases based on the number of failing test cases in a cluster. The formula for this adjustment can be seen in 3.1. L_i is defined as the new label for test case i. f_{ci} corresponds to the number of fails in the cluster of test case i and p_{ci} similarily corresponds to the number passes. The max value allows already failing test cases to keep their label and prevents them from being adjusted.

$$L_i = max(L_i, 1 - \sqrt{\frac{f_{ci}}{f_{ci} + p_{ci}}}) \tag{3-1}$$

After the clustering step, a Prinicipal Component Analysis (PCA) is used. However, this is not done to reduce the number of components, because the components are supposed to be ranked in the later steps. PCA is deployed in order to rotate the test cases along the axis of maximum variance. This means that every line of data is more expressive while not destroying the actual meaning of the different components. The PCA step is used together with the virtual test set appended to the data set, in order to apply the same rotation to the virtual set.

3.2 Refinement

Refinement for this project is clearly distinct to most other projects. Typical Machine Learning models are trained once and then used. In this project the model retrains for every available fault with a new virtual test set which is then evaluated. Thus, adjusting parameters were required to work for multiple models. This was especially difficult with the adjustment of epoch numbers. An evolution of the initial and one major intermediate result are displayed in Table 3-1.

Bug	Number Components	Initial Model	Intermediate Model
Chart_2	53	21	42
Chart_14	108	74	8
Chart_21	27	23	10
Chart_25	639	231	12
Time_1	414	117	9
Time_3	572	492	35
Time_5	448	224	42
Time_9	188	103	20
Time_18	461	291	229
Time_21	399	174	135
Time_23	132	65	12
Lang_30	11	2	1
Lang_32	19	13	7
Lang_36	22	5	3
Lang_63	12	12	1
Math_6	414	167	48
Math_8	22	14	7
Math_16	65	54	12
Math_38	56	23	10
Math_51	55	12	22
Math_64	24	23	8
Math_66	79	56	54
Math_71	73	48	28
Closure_3	1089	421	261
Closure_16	1751	1204	35
Closure_26	1940	989	125
Closure_27	17	5	3
Closure_37	540	232	345
Closure_43	1175	682	515
Closure_47	1046	253	365
Closure_49	1129	421	34
Closure_72	824	677	522
Closure_76	718	394	292
Closure_103	998	668	428
Closure_108	879	853	107

Table 3-1: Table of all bugs and their performance with the current model.

Name	Type	Description
Short/Long Model	boolean	Changes the epoch length of the model
Clustering	boolean	Enables clustering preprocessing
PCA	boolean	Enables PCA preprocessing

Table 3-2: The different setting available for the neural network and its preprocessing steps

The initial model was modeled after [5]. That model initially did not work, even when trying to copy it. Most results seemed random and did not localize the fault. The first major improvement was made using a weighted cross entropy with a stronger weight on recall, as well as reducing the batch size to 1. The batch size is counter intuitive since stochastic gradient descent should work better with minibatches, but this is not the case here. The original reference paper used a batch size of 10.

3.3 Implementation

The implementation was done following the original paper given by [5]. Additional steps, such as clustering, visualization and PCA were added afterwards. The benchmark models were added after the first results were acquired in order to compare them.

When applying clustering, the original idea was based on EM-Clustering. This was changed to Agglomerative Clustering due to the nature of the dataset. Since the dataset consists of only ones and zeros, the rank of the data matrix is lower than anticipated, allowing only one or two clusters for some faults when using EM-Clustering. Thus, the approach was changed such that it is not dependant on the rank of the data matrix.

In order to compare different models and different settings, the model is evaluated with multiple settings and preprocessing steps en- and disabled. The models are compared later on in Chapter 4. The setting variables are given in Table 3-2. All other settings are automated in order to adapt to different data spectra without requiring fine tuning for every fault.

The mechanism which defines the epoch length is complex. In order to dynamically set the epoch length based on the problem, additional values are introduced. in particular these values are the *average break* and the *current break value*. The average break (a_b) value is given by Equation 3.3. Variable l_i is the average loss of epoch i and k_j is a constant value depending on the model type (short/long). Value k_j is 800 for the long model and 500 for the short model.

$$a_b = l_i * k_j \tag{3-2}$$

The current break value is a cumulative sum that does not reset. It is given by Equation 3.3 and is designed such that it grows faster when the average break is either low or the epoch number is high. Variable c_b is the current break and variable i describes the current epoch number. Variable a_b is the previously defined average break. The cumulative sum is updated every five epochs. Once the current break value is higher than the average break value, the algorithm terminates.

$$c_b = np.log(1 + (i^4)/(a_b)) * i * 0.4/(a_b * *2) \tag{3-3}$$

4.1 Model Evaluation

Table 4-1 and 4-2 show the results for all chosen training and test cases. The parameters for all models were adjust using the training test cases. Mockito was only evaluated in the finalized version of each model. The different columns show the results of models using different preprocessing steps and combinations as well as different training lengths. The highlighted values correspond to the best first hit for a given test case using different models. Multiple models were chosen in order to reflect specific strengths and weaknesses of these models.

The results presented here are mostly reliable. However, the neural network is not deterministic, so the exact rankings can vary. In one single case, it was observed that for fault Chart25, the fault localization for the P_L model had a First Hit of 231, which is a clear outlier. That has not occured any other time.

The models using clustering without PCA (C_S, C_L), work better on the Maths fault than most other models. It was found that most of the Math problems are based on code additions rather than code removal. It is suspected that the clustering helps the model to localize faults of that type. PCA on the other hand does not have a specific type of fault. It helps localizing some faults (e.g. Time5), but also increases the first hit of other faults (e.g. Closure108).

The proposed models are capable of localizing most faults. However, their performance is split among the different models. There was no single model found that could successfully localize most faults. However, the result seen here is promising. More often than not, some faults are difficult to localize for most models. This is to some degree also reflected here. None the less, for almost all faults, at least one model performs reasonably well. The result presented here performed worse overall than anticipated, but instead localized faults, that are typically difficult to. When taking the number of components into account, the worst best localization is given by Mockito25, where a bit more than 50% of all components are suspected before the faulty component.

4.2 Justification

The tarantula benchmark model has the highest count of lowest *First Hits* of all models presented. Ochiai only performs well on a select few problems that are difficult for most

Buggy version	NC	S	L	P_S	P_L	C_S	C_L	CP_S	CP_L	T	O
Chart_2	53	25	32	39	9	17	20	11	42	36	43
Chart_14	108	6	10	7	47	6	13	14	13	8	97
Chart_21	27	3	8	4	3	3	1	2	2	2	14
Chart_25	639	299	419	2	3	5	129	4	63	5	631
Time_1	414	91	8	2	87	64	27	95	3	2	387
Time_3	572	2	18	1	1	4	20	1	1	1	431
Time_5	448	46	118	26	10	82	59	11	26	11	372
Time_9	188	73	62	26	43	38	31	76	26	16	164
Time_18	461	204	315	264	348	223	364	155	262	10	443
Time_21	399	24	4	137	149	108	38	82	171	4	352
Lang_30	11	2	1	1	1	1	1	1	1	3	4
Lang_32	19	5	1	2	3	2	6	6	9	8	10
Lang_36	22	3	3	8	6	7	2	7	7	1	6
Lang_63	12	2	1	4	2	2	2	2	4	2	6
Math_6	414	38	22	127	40	110	15	54	77	27	12
Math_8	22	2	10	2	2	1	2	1	1	1	11
Math_16	65	1	26	3	11	50	8	14	4	1	33
Math_38	56	22	54	17	24	9	31	20	11	19	28
Math_51	55	48	30	20	34	33	12	40	12	30	28
Math_64	24	18	15	12	13	11	16	5	11	7	12
Math_66	79	33	17	41	24	17	5	26	15	62	9
Math_71	73	57	35	62	47	39	8	55	62	43	62
Closure_3	1089	54	5	59	241	299	410	156	360	68	538
Closure_16	1751	1	1	5	1	1	9	1	5	50	358
Closure_26	1940	330	39	12	7	1084	34	16	9	4	329
Closure_27	17	1	1	6	2	1	1	11	2	2	9
Closure_37	540	226	265	247	24	348	166	80	376	227	270
Closure_43	1175	159	832	225	446	698	661	1125	636	405	572
Closure_47	1046	1	318	308	27	217	18	17	487	19	399
Closure_49	1129	273	209	45	279	158	628	388	58	422	271
Closure_72	824	380	562	85	61	581	63	334	62	35	412
Closure_76	718	11	321	93	21	15	287	12	21	18	341
Closure_103	998	266	88	124	181	93	49	52	160	57	176
Closure_108	879	119	18	224	193	280	44	65	153	39	440

Table 4-1: Results of all models on the training set. NC: Number of Components, L: Long running Model, S: Short running Model, C: Model uses Clustering, P: Model uses PCA, T: Tarantula, O: Ochiai

Buggy version	NC	S	L	P_S	P_L	C_S	C_L	CP_S	CP_L	T	O
Mockito_1	441	5	5	5	5	5	11	5	6	105	78
Mockito_2	184	11	53	2	11	1	9	9	5	13	10
Mockito_3	369	56	317	61	143	215	336	77	73	88	255
Mockito_4	329	52	1	5	1	80	2	36	4	1	255
Mockito_6	399	14	1	9	1	18	1	11	11	2	321
Mockito_7	247	7	114	150	12	178	80	18	156	12	124
Mockito_8	16	10	6	14	5	10	14	14	13	14	8
Mockito_9	305	1	60	3	30	14	16	58	3	11	232
Mockito_10	260	1	59	38	8	108	43	29	39	10	219
Mockito_11	132	11	1	1	42	20	1	24	1	1	95
Mockito_12	107	21	2	1	24	20	2	21	2	20	1
Mockito_13	263	27	151	249	132	125	40	149	257	128	92
Mockito_14	224	34	43	157	44	20	35	59	143	49	112
Mockito_15	163	88	134	149	118	94	150	122	132	82	85
Mockito_16	162	117	67	97	20	32	109	39	118	82	81
Mockito_17	83	47	2	1	12	19	2	5	1	1	42
Mockito_18	138	66	1	4	21	83	4	39	1	1	53
Mockito_19	301	22	2	135	9	5	5	22	83	20	151
Mockito_20	328	181	144	100	270	139	86	166	156	46	89
Mockito_21	92	2	1	2	3	3	10	3	2	2	46
Mockito_22	127	89	3	5	20	38	5	4	5	7	42
Mockito_23	309	32	96	128	41	8	55	45	93	14	258
Mockito_24	378	186	288	376	283	281	319	142	376	84	233
Mockito_25	376	7	10	44	111	5	20	23	25	18	191
Mockito_27	247	47	80	48	18	111	96	31	36	9	218
Mockito_28	191	136	127	155	100	85	81	172	183	79	96
Mockito_29	228	22	132	37	21	7	171	1	7	7	114
Mockito_30	125	5	5	1	11	17	4	4	1	1	100
Mockito_31	121	1	2	4	13	13	1	7	4	1	95
Mockito_32	272	271	177	53	250	271	178	258	54	245	102
Mockito_33	257	27	12	30	7	127	7	30	23	46	76
Mockito_34	208	149	24	35	103	141	32	52	34	43	76
Mockito_35	170	23	12	2	2	28	7	10	2	1	164
Mockito_36	181	38	1	1	30	74	1	29	1	1	46
Mockito_37	161	9	1	6	6	3	4	3	9	4	47
Mockito_38	208	5	32	7	10	5	1	8	5	4	17

Table 4-2: Results of all models on the test set. NC: Number of Components, L: Long running Model, S: Short running Model, C: Model uses Clustering, P: Model uses PCA, T: Tarantula, O: Ochiai

Train/Test	S	L	P_S	P_L	C_S	C_L	CP_S	CP_L	T	O
Train S_A	63.92	94.4	53	56.17	111.79	77.21	68.13	72.5	39.83	190.44
Train NS_A	0.21	0.27	0.21	0.19	0.26	0.2	0.21	0.21	0.16	0.54
Test S_A	71.68	68.32	82.32	74.14	74.73	64.18	63.36	78.91	44.77	106.95
Test NS_A	0.34	0.28	0.33	0.30	0.33	0.27	0.27	0.32	0.2	0.49

Table 4-3: Summed results for training and test set. S_A: Average Sum of First Hits, NS_A: Normalized Sum of First Hits, L: Long running Model, S: Short running Model, C: Model uses Clustering, P: Model uses PCA, T: Tarantula, O: Ochiai

other models to locate. This is also reflected by the results given here. The difference between the benchmark and the constructed models in their performance is similar in both training and test set. Compared to any single proposed model, tarantula performs better. This is reflected by the average and normalized average first hit as seen in Table 4-3. Of the proposed models, the overall performance on the training set is best with the PCA only models. For the test set, the models CP_S and C_L perform best.

In Figure 4-1 and 4-2 the respective ranking over time for the faults Mockito25 and Mockito31 can be seen. One timestep on the x-Axis is equal to five epochs. The examples chosen display the more frequent and different behaviors of the component rankings. These images were taken from the long model with clustering and PCA. In the left image it can be seen that some components are localizable quite quickly, but loses their better ranking later on in the training step. This mostly happens when the faulty component seems obvious at first glance. In other words, when the neural network still is simple, it is capable of localizing the fault, but a more complex approach looses sight of the simple faults. The short model was introduced to alleviate this problem. The right image shows the most common ranking over time. The fault starts unlocalized and is localized over time. However, in most cases the improvement flattens and running for additional time would not reduce the first hit of the fault.

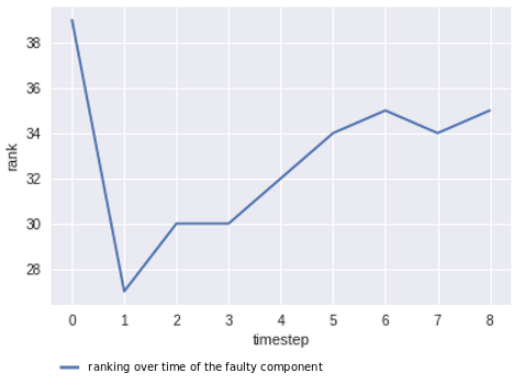

Figure 4-1: Ranking over time for the Components of Mockito_25.

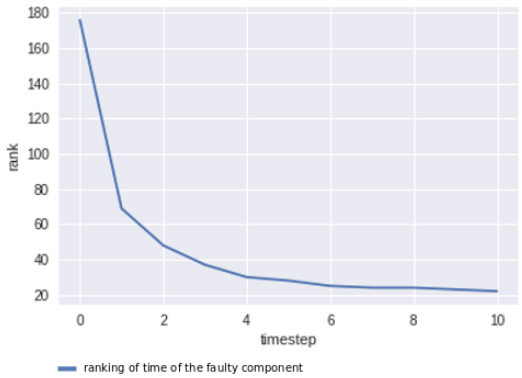

Figure 4-2: Ranking over time for the Components of Mockito_31.

5.1 Free Form Visualization

Train/Test	S	L	P_S	P_L	C_S	C_L	CP_S	CP_L	T	O
Train	7	8	5	6	6	7	5	4	10	1
Test	7	11	7	6	5	6	2	5	13	1

Table 5-1: Sum of best first hits (green cells of Table 4-1 and **??**

Table 5-1 shows the number of best first hits any given model has. While tarantula has the most, the other models also have a non-neglectable amount of best first hits. This shows that all models and their different settings have both weaknesses and strengths. Because of this property, the models are primed to be used for ensemble methods. This is further explained in Section 5.3.

Additionally, the capability of localizing some difficult fault should be demonstrated. The code given in Figure 5-1 is taken from the Mockito1 fault. Both benchmark models perform bad with a rank higher than the top 100. However, almost all proposed models have this fault in the top 10 (exception model C_L, rank:11). The fault is difficult due to the lack of information from the exception as it was simply unsupported. Additionally the problem is fixed using code additions, which is generally considered more difficult to localize.

13 projects/Mockito/1/org/mockito/internal/invocation/InvocationMatcher.java View

```
@@ -120,7 +120,18 @@ public Location getLocation() {
120 120     public void captureArgumentsFrom(Invocation invocation) {
121 121         if (invocation.getMethod().isVarArgs()) {
122 122             int indexOfVararg = invocation.getRawArguments().length - 1;
123     -           throw new UnsupportedOperationException();
    123 +           for (int position = 0; position < indexOfVararg; position++) {
    124 +               Matcher m = matchers.get(position);
    125 +               if (m instanceof CapturesArguments) {
    126 +                   ((CapturesArguments) m).captureFrom(invocation.getArgumentAt(position, Object.class));
    127 +               }
    128 +           }
    129 +           for (int position = indexOfVararg; position < matchers.size(); position++) {
    130 +               Matcher m = matchers.get(position);
    131 +               if (m instanceof CapturesArguments) {
    132 +                   ((CapturesArguments) m).captureFrom(invocation.getRawArguments()[position - indexOfVararg]);
    133 +               }
    134 +           }
124 135
125 136         } else {
126 137             for (int position = 0; position < matchers.size(); position++) {
```

Figure 5-1: The Mockito 1 fault of the Defects4j dataset as found in [1]. The code lines indicated by color display the required changes to solve the fault.

5.2 Reflection

Writing the program was very difficult in hindsight. I did not have much knowledge in Software Testing beforehand, which made the project even more difficult. Although interesting, the chosen approach was complex to set in motion, because it retrains a new neural network for every fault. Automatically adjusting the parameters for all given training faults was difficult and especially tedious. However, dynamically adjusting epoch lengths was an interesting experience.

The final model has interesting consequences. While its current form should not be used in a general setting of that type, the model solves and localizes some faults, that I have not seen localized previously in any of the papers read during the project. There definitely is some value in the models proposed, although they need some polishing. Section 5.3 contains more information on that.

5.3 Improvement

As previously mentioned, the different models are capable of localizing almost any faults and often do not share which faults they localize well. This could be alleviated using some sort of ensemble. For example, a combined model as an ensemble where the suspiciousness values of all models are normalized and averaged in order to get a final suspiciousness value could be used. Of course more complex ensembles would also be possible, where different models have weights added to them in order to localize a wider variety of faults with a lower first hit. Other improvements could for example include component reduction with feature selection as a preprocessing step.

Another approach that was specifically looked at is described in [2]. It is based on Radial Basis Function (RBF) Neural Networks. The approach however seemed less unique although more difficult than the one chosen for this project. Although outside the scope of the project, an interesting approach would have been combining these two.

Bibliography

[1] Defects4j Presentation Urls. https://github.com/Spirals-Team/defects4j-presentation-urls. Accessed: 2018-07-01.

[2] Eric Wong, W.; Debroy, V.; Golden, R.; Xu, X.; Thuraisingham, B.: Effective Software Fault Localization Using an RBF Neural Network. In: vol. 61 (Mar. 2012), pp. 149–169.

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

[4] 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.

[5] 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.