**COMP 3211 Software Engineering**

**Project Report**

**Group 23:**

**Ho Tin Yau              15070597D**

**SARTAYEVA Yerkezhan     16094821D**

**ZHANG Ziwei 15103037D**

# Tarantula

## Theory

Debugging is one of the most time-consuming tasks in software development. Therefore, it is no surprise extensive research is conducted to design effective fault-localisation tools. An example of such a tool is the Tarantula fault-localisation algorithm designed by [1]. The main rationale behind the algorithm is that the more a statement is used by failed test cases (with wrong output), the more likely it is to be faulty. The algorithm evaluates the frequency of coverage for each statement s in test cases to obtain the suspicion score of s, which reflects the likelihood of s to contain a bug. Here is the formula developed by the authors to calculate the suspiciousness score of each statement in code:

(1.1)

Please see the table below for variable descriptions.

|  |  |
| --- | --- |
| **Variable** | **Description** |
| s | A statement in the source code |
| passed(s) | The number of passed test cases (with correct output) that covered s |
| totalpassed | The total number of passed test cases |
| failed(s) | The number of failed test cases (with incorrect output) that covered s |
| totalfailed | The total number of passed test cases |
| hue(s) | Hue ranges between 0 and 1. A lower hue indicates that s it is not covered by many passed cases. Therefore, the lower the hue, the higher the suspiciousness. |

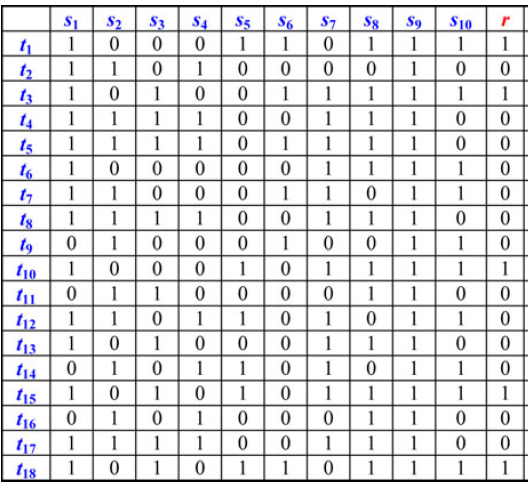
Table 1 Variable Descriptions

The suspiciousness scores of each statement based on multiple test cases are recorded in an array and sorted in descending order. The statement with the highest suspiciousness score is most likely to contain a bug, so it should be checked first.

## Implementation

### Input

To stay consistent with other algorithms, the coverage matrix from [3] (Crosstab) was used for input representation. The Parser class reads data from HTML coverage files and converts it into a coverage matrix m, where m[t][s] = 1 means statement s + 1 is covered in a test case t + 1 (+1 because indices start with 0 in code). Rows in the coverage matrix correspond to each test case, and columns correspond to each statement. The length of one row r in the coverage matrix is s + 1, where the first s values are statement coverage bits and the last value is the outcome of test case r (1 if test case t fails and 0 otherwise). See the picture below for a sample coverage matrix from [3] (Crosstab).

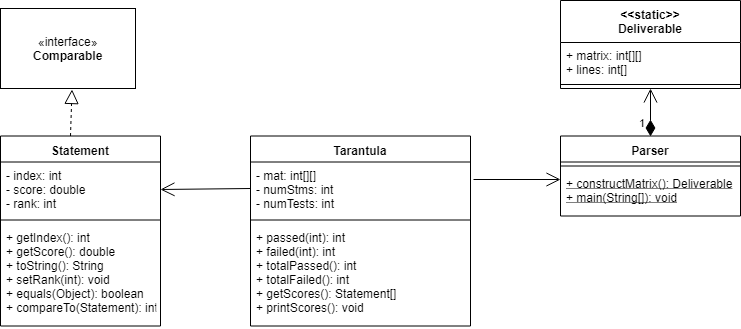


Picture 1 Sample Coverage Matrix

Parser both returns a coverage matrix as an array object and writes the matrix to a file called “matrix.txt”. Tarantula accepts the matrix generated by Parser as a 2D array object instead of reading from a file.

### Implementation

A class Tarantula is created to hide all auxiliary methods from the user. The class accepts a 2D integer matrix m, whose format is described above. 4 methods are created for to calculate passed(s), failed(s), totalpassed and totalfailed. The getScores() method is the main method that returns an array of Statement objects with calculated suspicion scores and ranks. See the UML diagram below for class relationships:



Picture 2 UML Class Diagram of the Tarantula Module

The Tarantula class does not have any static methods or attributes, which means it needs to be instantiated first. This was done for 2 reasons:

1. to make sure the matrix is not passed in every single method as a parameter; the matrix is copied to Tarantula’s matrix attribute and used throughout the class.
2. getScores() is the main method that uses all other methods. If it is made static, then all other methods must be static. Static methods cannot use non-static attributes, so all attributes will have to be made static as well. Static attributes are shared among all instances of a class. If the user wants to compare the output of 2 matrices, he/she will not be able to do so in a single run as all objects will reference the same matrix.

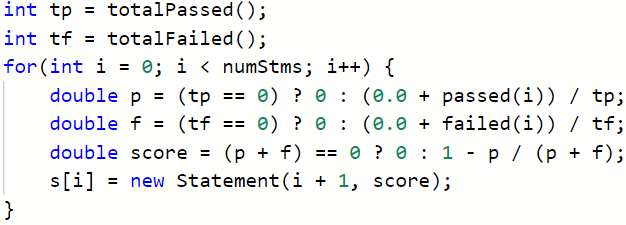
This is not to say that this approach is the correct one; it was adopted for personal convenience.

To store 3 items of data per statement together, a Statement class was implemented. The Statement class implements Java’s Comparable interface to make sure an array of Statements is sortable. The description of the main getScores() method follows.

First, getScores() method creates an auxiliary array of Statements to store their data. Then, suspiciousness scores are calculated in the main loop and stored in the array. Suspicion scores are precise to 6 decimal places, i.e. 6.0000001 = 6.000002. To prevent division by 0, fractions with denominator = 0 are set to 0, as was recommended in the paper. This assumption, however, is an example of binary thinking: a statement can be either used by faulty or passed test cases, but the third, neutral option of a statement not being covered by any test case is not catered for. To elaborate, there are 2 reasons why hue(s) = 0:

1. s has a bug because it is only used by faulty test cases
2. s is never covered, i.e. it is unreachable, e.g. an unreachable return statement

In this case, if either passed(s) + failed(s) = totalCoverage(s) = 0, then hue(s) will be set to 0, which is wrong. Some programs may unintentionally contain lines that are never covered, e.g. an if statement whose condition never evaluates to true. These lines tell us nothing about the location of the bug. Therefore, they are harmless, i.e. even though they were never covered, their hue should not be red. The hue should not indicate coverage but should indicate the likelihood that a statement is harmless. In this case, statements that are not covered should have a hue of 1, i.e. green. Here are lines that cater for these special cases:



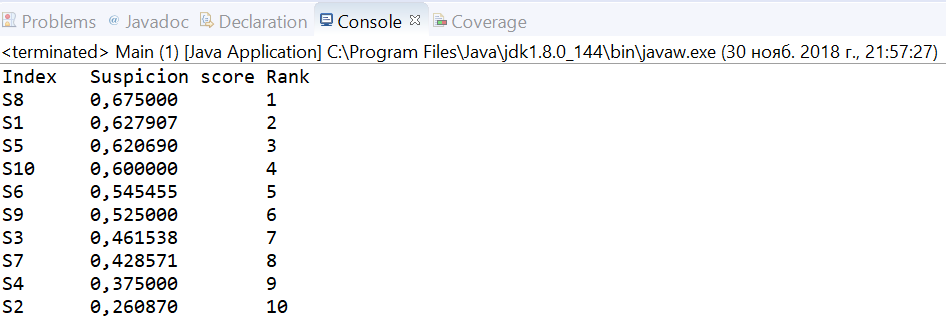
Picture 3 Handling Special Cases

Nevertheless, the formula later in the paper (see equation 1.3 below) seems to handle the special case of totalCoverage(s) = 0 because if failed(s) = 0, then the suspiciousness score is set to 0 regardless of why failed(s) = 0. Even if the denominator is 0 (which reflects total coverage), the suspiciousness is 0, which is correct. However, this formula still does not solve the problem of hue(s) being 0 when coverage is 0.

After suspicion scores are calculated, statements are ranked and sorted in descending order of suspiciousness. Ranking is done in the same way as was suggested in the paper, i.e. statements with equal suspiciousness have the same ranking.

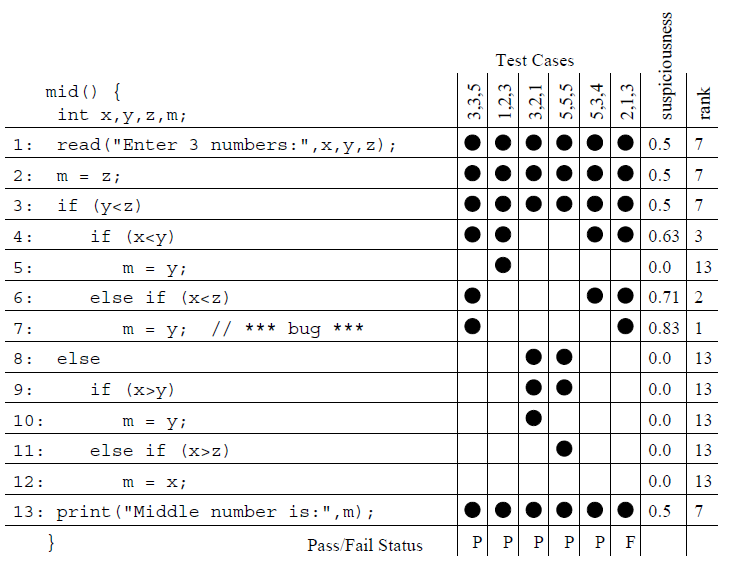
### Output

Here is the screenshot of the final output for a matrix from [3] (Crosstab, page 44):



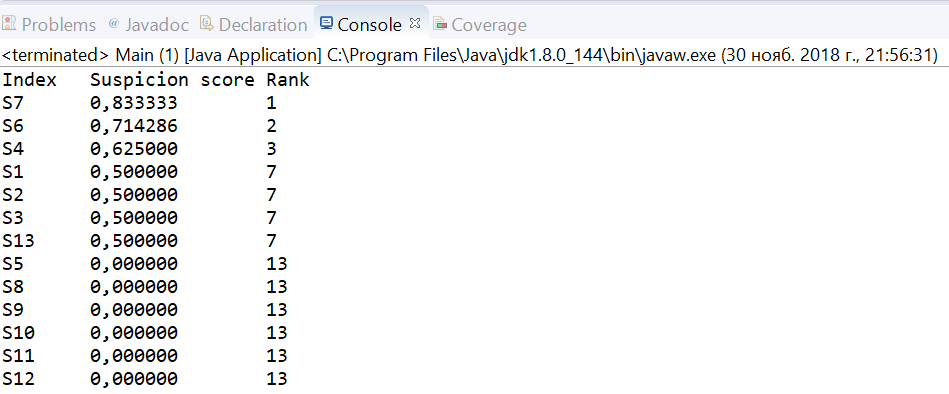
Picture 4 Program Output for Coverage Matrix in [3] (Crosstab)

Page 45 in [3] (Crosstab) lists the order of the statements: (8, 1, 5, 10, 9, 6, 3, 7, 4, 2)[1 10, 9, 6, 3, 7, 4, 2], which is almost the same as the output except that lines 9 and 6 are reversed. This could be because their suspiciousness scores are close. To test whether the program produced the same output as that of a sample matrix in [1] (Tarantula), a coverage matrix was constructed manually based on this representation from [1]:



Picture 5 The Coverage Matrix of a Sample Program from [1] (Tarantula)

From Picture 5, the ranking is (7, 6, 4, 1, 2, 3, 13, 5, 8, 9, 10, 11, 12). Here is the screenshot of the program’s output:



Picture 6 Program Output for Matrix in [1] (Tarantula)

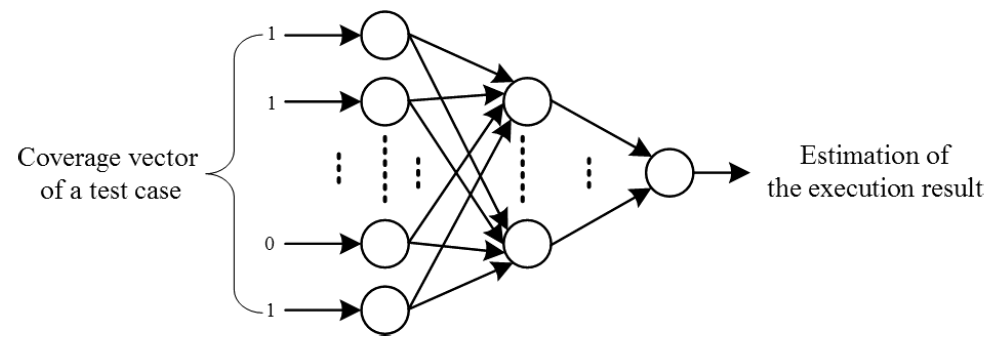
In this case, the rankings, suspicion scores and the ordering are identical to those in Picture 5, which suggests the program runs correctly.

# Backpropagation Neural Network

## Theory

### Background

[2] propose to use a back-propagation neural network (BPNN) for fault localisation, which refers to a machine learning model that learns the relationship between input and output. They first construct a neural network with 3 layers (input, hidden and output) as shown below.



Picture 7 BPNN Model in [2]

BPNN is based on the assumption that there is a relationship between input and output and that given x, y can be predicted after learning function that maps input to output. The input layer accepts input values x and calculates output values according to its function (forward propagation). Then the model compares its output with correct output values y, computes the error and adjusts its weights to make future predictions more accurate (back propagation). BPNN is powerful because it has many applications and can approximate non-linear functions.

### Training

The authors’ neural network has m input nodes, where m is the number of statements, 3 hidden nodes and 1 output node. The sigmoid function is chosen for transfer functions. The model accepts coverage vectors of test cases one by one and calculates the execution result, which ranges between 0 and 1 (the higher the value, the more likely the test case to fail). A coverage vector is the coverage of statements in a single test case, i.e. it corresponds to a single row in a coverage matrix.

|  |  |
| --- | --- |
| **Notation** | **Meaning** |
| x | Input matrix |
| w | Weight matrix |
| bh | Hidden layer’s bias matrix |
| bo | Output layer’s bias matrix |
|  | Input-to-hidden weights |
|  | Hidden-to-output weights |
| y | Target data matrix (expected output) |
| J | cost |
|  | Learning rate |

Table 2 Explanation of the notations used in the report

1. Forward propagation: First, the output of each hidden node hi is calculated with the following formula:

(2.1)

(1.5)

Equation 2.1 is a sigmoid activation function. Once this step is complete, output layer values are calculated as follows:

(2.2)

is the output that indicates how likely a test case to be failed.

1. Back propagation: The main purpose of back propagation is to adjust weights and biases to polish the objective function. Adjustments must be done for thousands of times until error is minimal (0.001 in the paper). Here are formulas required for backpropagation:

|  |  |  |
| --- | --- | --- |
| **Backpropagation Step** | **Formula** | **Equation #** |
| Total output error |  |  |
| Backpropagation error for output |  |  |
| Gradient descent with respect to |  |  |
| Gradient descent with respect to |  |  |
| Backpropagation error for hidden layer |  |  |
| Gradient descent with respect to |  |  |
| Gradient descent with respect to |  |  |
| Update |  |  |
| Update |  |  |
| Update |  |  |
| Update |  |  |

Table 3 Backpropagation Steps

The main objective is to adjust the weights and biases so that in the next forward pass, the output y is more accurate. We need to propagate forward and backward for thousands of times and keep reducing the cost (error).

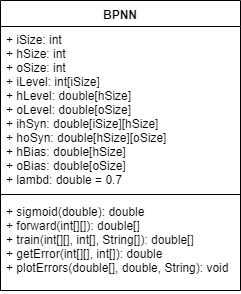
## Implementation

### Input

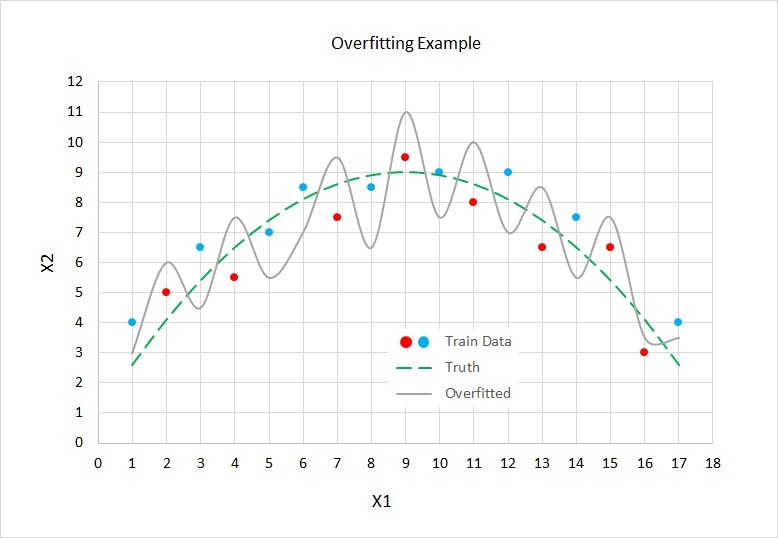
Python was chosen to implement a BPNN. Parser is implemented in Java, so it cannot directly be used in Python code. Instead, the Python program reads from a text file generated by Parser. The matrix in the file is in the same format as that in Tarantula, but it is separated in the source code. The outcomes of test cases (failed or passed) are extracted into a separate 1D array (matrix of target values y), while statement coverage bits are stored in another 2D array (matrix of input values x). The input matrix is of size , where n is the number of test cases and m is the number of statements. The output matrix’s dimensions are , i.e. 1 output for each test case.

### Implementation

BPNN calculations have to be efficient given that the neural network must be trained for thousands of times. For that reason, NumPy library is used. Its efficient array and math operations help speed up execution. Matplotlib library is used to display the learning rate of the model as a graph. A class is created to represent a neural network. See the picture below for the description of the class:



The model has the same parameters as that in [2] except that L2 regularisation is used to pre-empt over-fitting as is illustrated below.



Picture 8 Overfitting Example [3]

According to [3], L2 regularisation can be incorporated as follows:

(2.12)

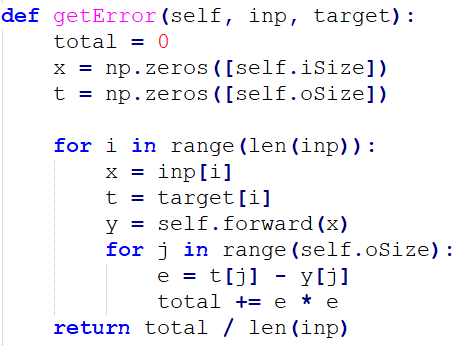
Most model parameters were taken from [2], but some were derived from external literature. See the table below for model parameters and their values.

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| Number of input nodes (iSize in code) | m = number of statements in the input matrix |
| Number of hidden nodes (hSize in code) | 5, as was suggested in [2] |
| Number of hidden layers | 1 |
| Number of output nodes | 1 |
| Learning rate | 0.5 (derived experimentally) |
| Number of trainings | 6000 |
| for L2 regularisation | 0.005, as was suggested in [3] |
| Activation function | Sigmoid |

Table 4 Model Parameters

Initially, weight and bias values are set to random values from 0 to 1. Once model parameters are defined, training should be carried out. In each of the 6000 trainings, for each test case t, the model does the following:

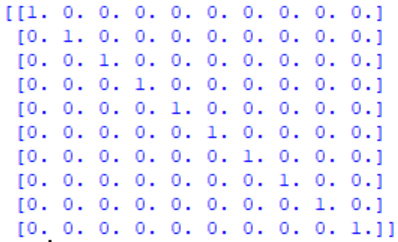
1. Propagate forward with the coverage vector x[t] and the target output y[t]
2. Calculate gradient descents for weights and
3. Calculate bias gradients and
4. Update weights and by subtracting the error
5. Update biases and by subtracting the error
6. Every 1000th training, calculate and print the average error for all test cases by propagating forward. Here is the screenshot of the getError() function:



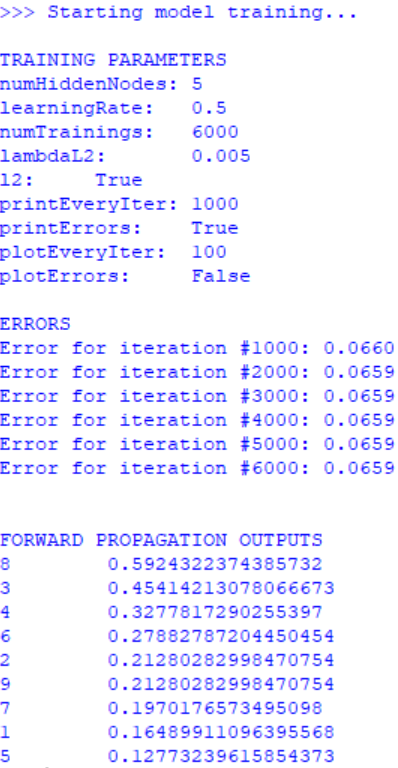
Picture 9 Calculating Model Error

### Output

Once training is complete, suspicion scores of statements in input matrix x are calculated by propagating forward with an identity matrix as shown below:

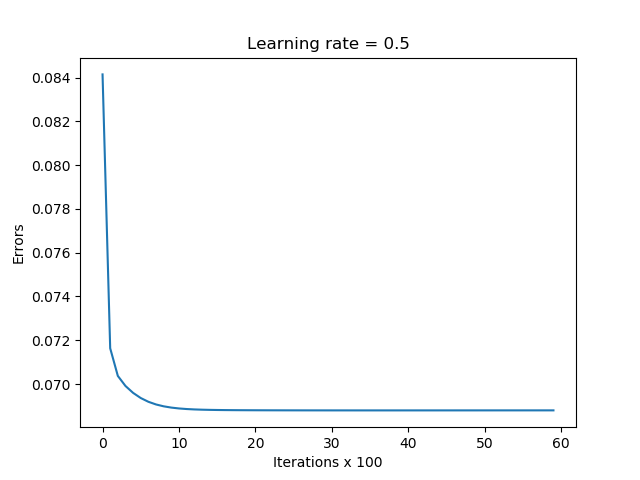


This method was proposed by [2]. The identity matrix is of size . Here are outputs for Matrix 3 with L2 regularisation (see the Appendix):



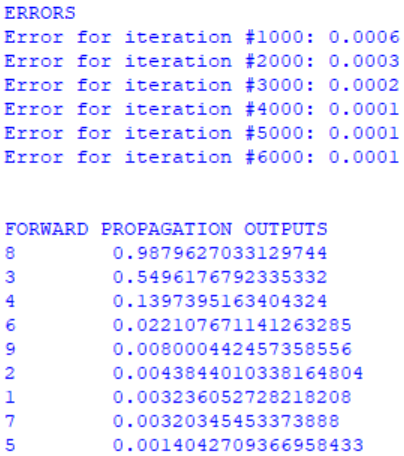
Picture 10 Program Output for Matrix 3 with L2 Regularisation

Increasing the number of trainings does not change the error significantly, so it is capped at 6,000. As can be seen from Picture 10, the model is only 59% confident the 8th statement contains a bug (the 8th statement actually has a bug), so it is not very good at predicting suspiciousness scores. Here is a graph representing the learning process of the model:



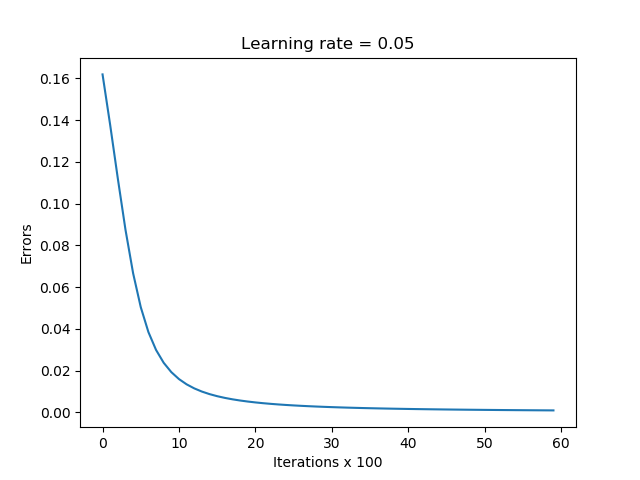
Picture 11 Model Learning

However, without L2 regularisation, the model becomes much more confident about its predictions as is shown below:



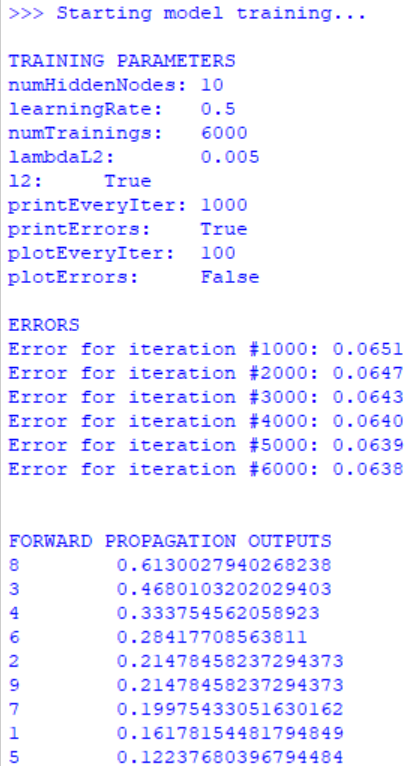
Picture 12 Program Output for Matrix 3 with No L2 Regularisation

The final error is 0.0001. Learning is smoother when learning rate is low:



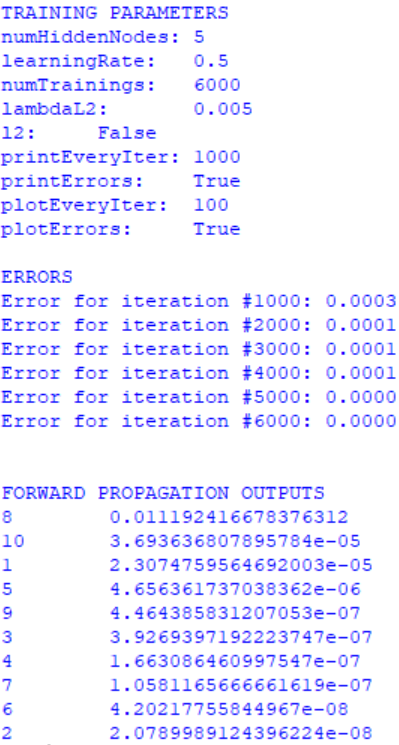
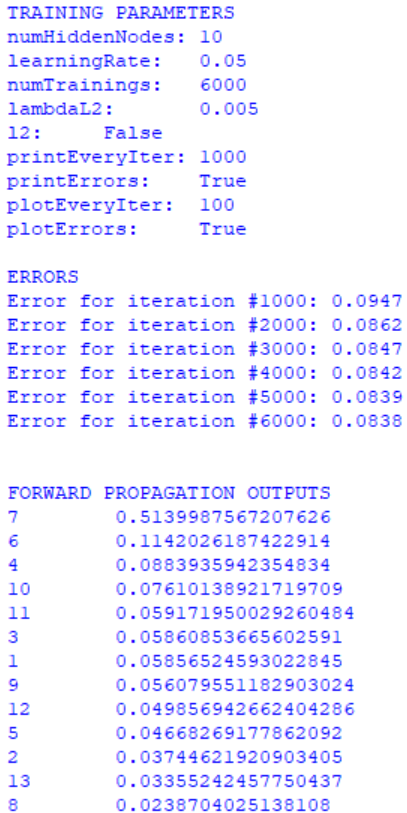
Picture 13 Model Learning

Increasing the number of hidden layers improves the correctness of the output with L2 regularisation:



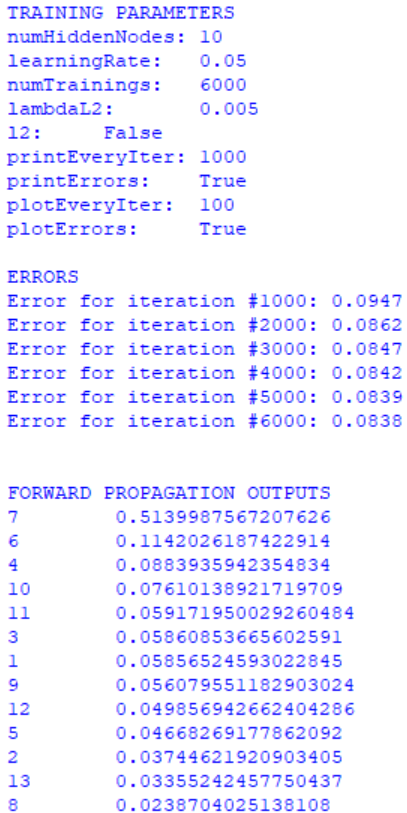
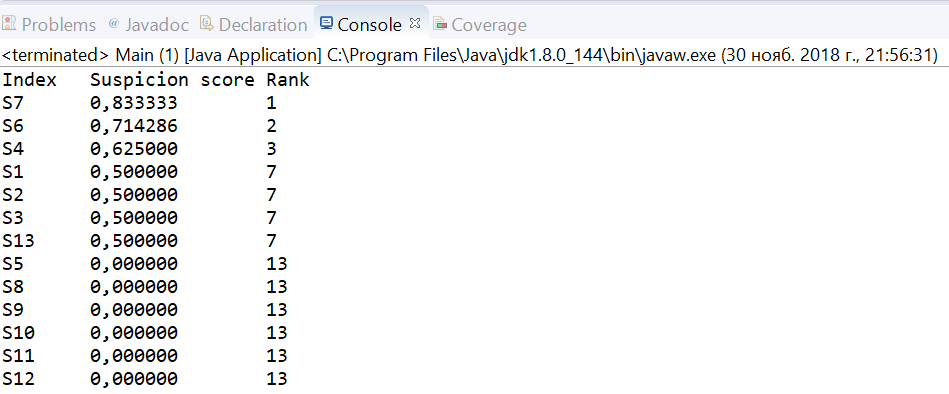
Picture 14 Program Output for Matrix 3 with L2 Regularisation and Number of Hidden Nodes = 10

To check if the ranking of the statements is correct, the model is run with Matrix 1 and Matrix 2, whose correct orderings are known (see the Appendix):



Picture 15 Left: Output for Matrix 1; Right: Output for Matrix 2

It is evident that the model is not confident about its outputs, but the error is low.



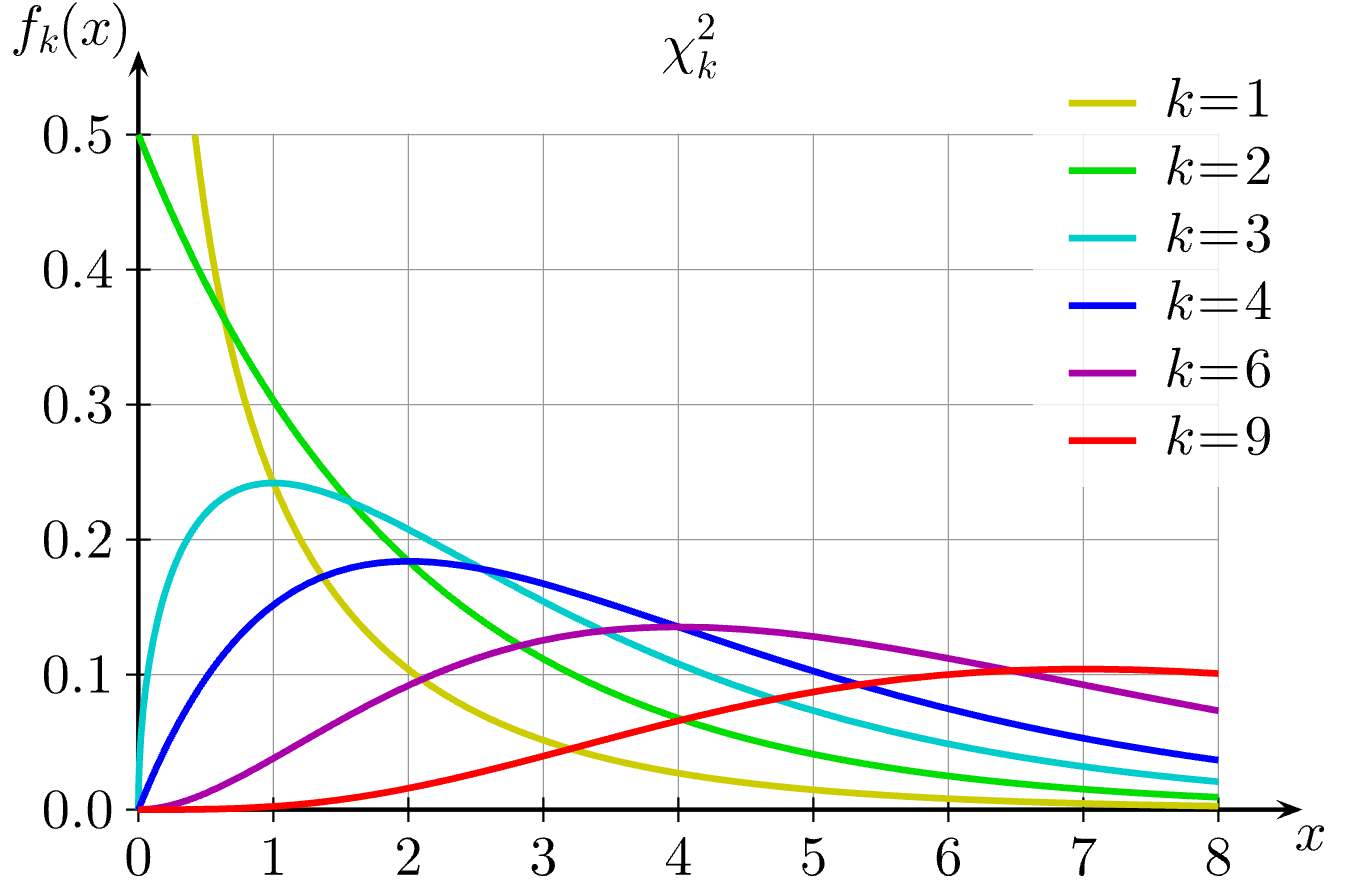
Picture 16 Comparison of Tarantula's Output with BPNN

Comparing the output with Tarantula, the first 3 statement’s ranking is correct, and statements 1, 2, 3, 13 and 5, 8, 9, 10, 11, 12 have the same scores. The faulty statement was identified correctly in all 3 matrices, but the ordering of the remaining statements is not necessarily the same as that of Tarantula.

# Crosstab

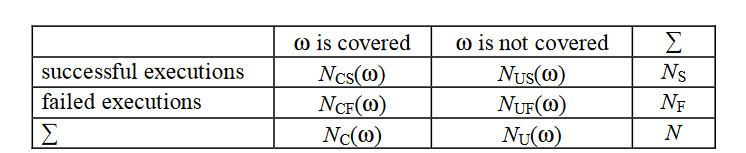
## Theory

The crosstab method proposes that there is a crosstab for every statement in the program. The aim of the method is to rank every statement based on its suspiciousness and let the programmers examine the faultiest ones first. The assumption is that χ2(ω) follows the Chi-Square distribution and the null hypothesis is that the result of passing or failing the test case is not related to the statement.



*Picture 17 The Chi-Square Distribution [7]*

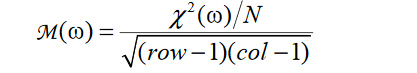
The crosstab for each statement consists of nine correlated elements. The chi-square formula is given in Picture 19. To eliminate the impact of the sample size, the contingency coefficient is calculated. The row and col are both 2 in our case. Then the last step is to calculate the suspiciousness of each statement and rank them from largest to smallest.



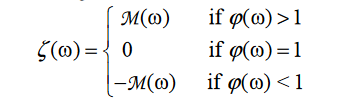
*Picture 18 The Crosstab for a Statement [4]*



*Picture 19 The Chi-square statistics [4]*



*Picture 20 The Contingency Coefficient [4]*



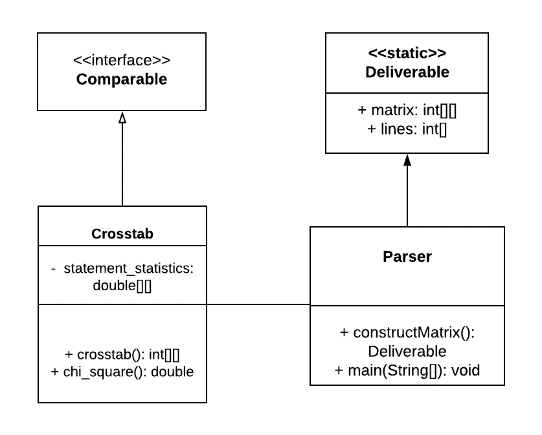
*Picture 21 The Suspiciousness [4]*

## Implementation

*Input*

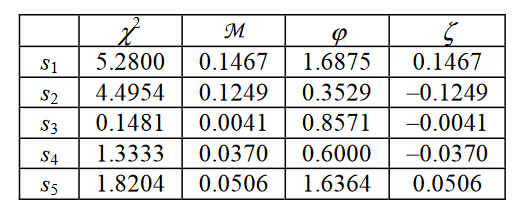
The input to the crosstab program has exactly the same format as input to the Tarantula program described in section 1.2.

*Implementation*



*Picture 22 UML Class Diagram of the Crosstab Module*

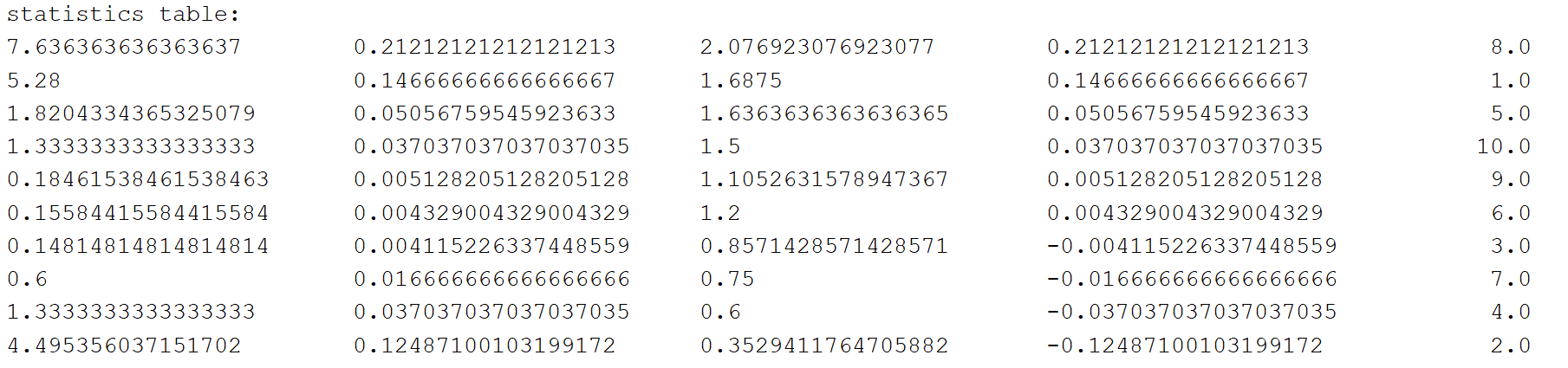
The crosstab of each statement is calculated and then the crosstab is passed to the function chi\_square used to generate the Chi-square. The statement\_statistics matrix has the size no. of statements \* 5, which contains the four attributes shown in the below table as well as the line number in the last column. The line number is to keep track of the line after sorting.



*Picture 23 Statement Statistics [4]*

At last, the comparable interface will take the statement\_statistics and sort it by the fourth column, which is the suspiciousness.

The output statistics table after sorting are shown below:

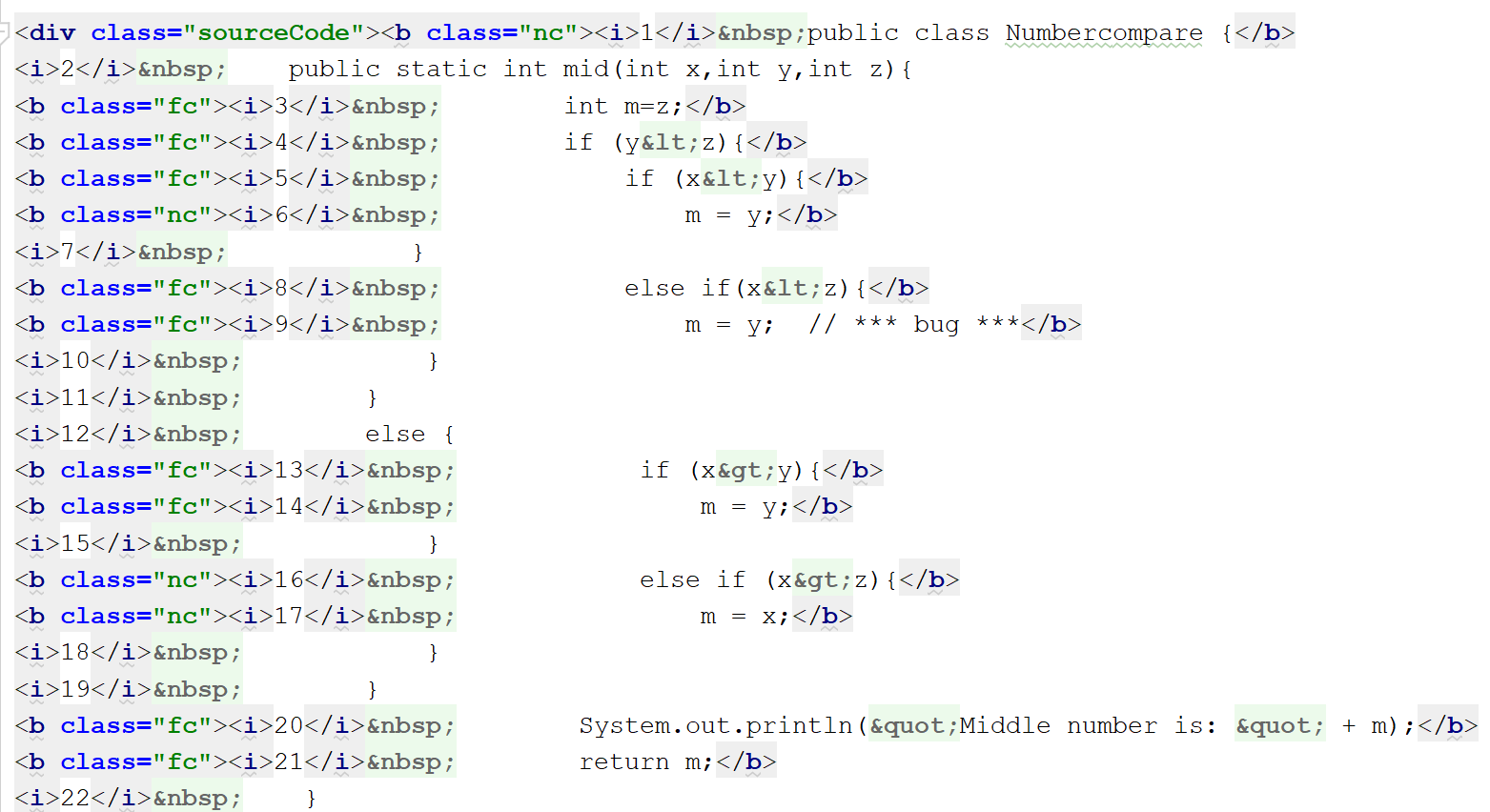


*Picture 24 The Sample Output of Crosstab*

# Generating coverage matrix

The test cases are written using JUnit and after running the test cases with coverage, some coverage reports in .html will be generated. The lines with the class label of “fc” indicate covered lines and those with the class label of “nc” indicates uncovered lines. For the lines without a class label, they are not considered as a statement. The passed test cases are denoted by adding the character “p” in the class label.

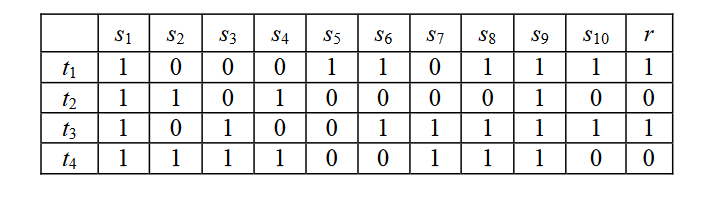
Below shows a part of the test coverage result of a failed test case of the “Number Compare” program similar to the one in the Tarantula paper.



*Picture 18 The HTML Test Coverage*

Parser.py parses through these coverage HTML files and returns the coverage matrix. Apart from the coverage matrix, Parser.py also returns a list of the line numbers of the program. Because as some symbols such as “}” occupies one line, but they are not counted as a statement, their line numbers are skipped. In the above example, the list of the line numbers is 1, 3, 4, 5, 6.

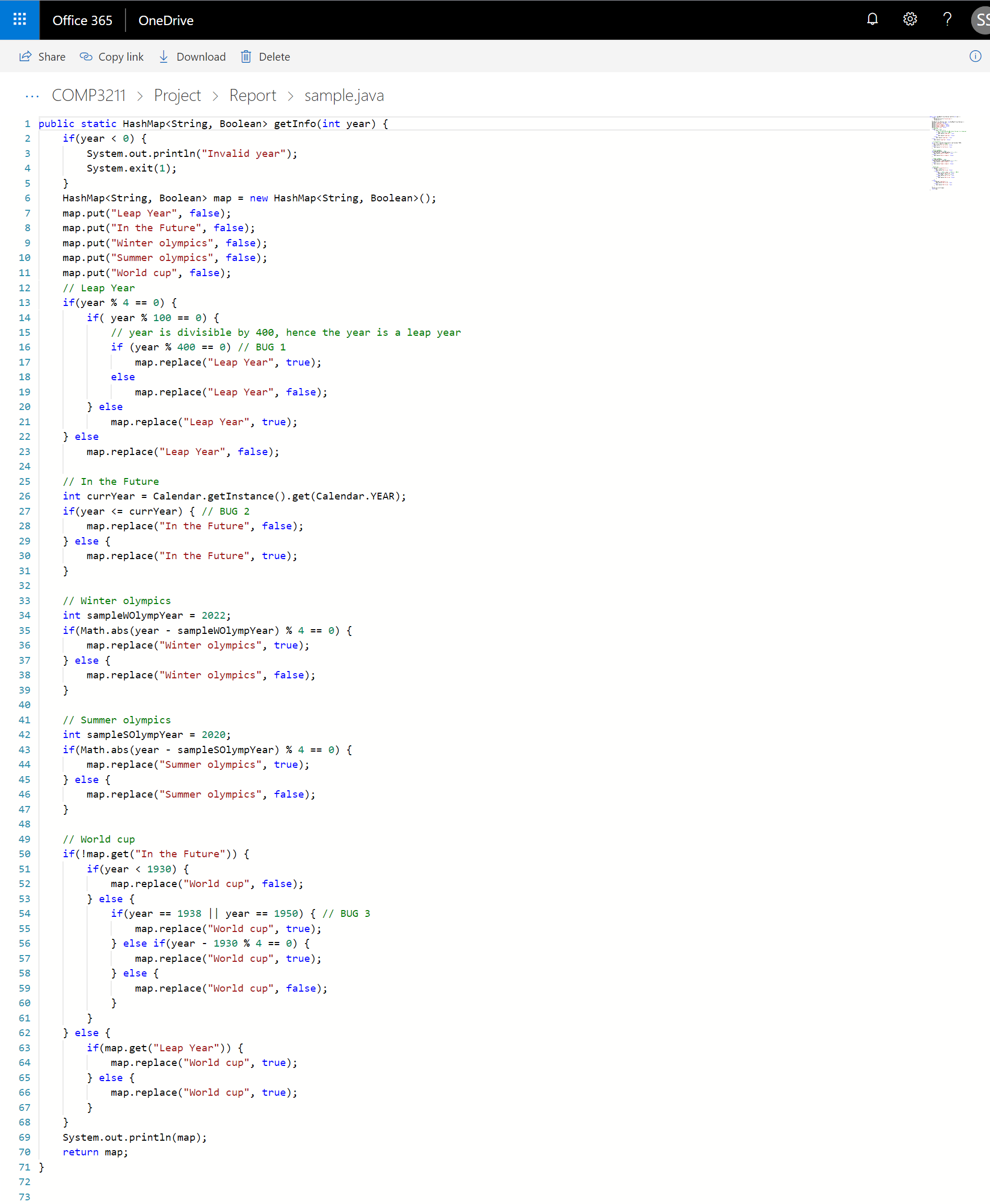
The coverage matrix has the size of no. of test cases \* (no. of statements + 1), with the last column denoting the passing or failing of the test case.



*Picture 19 A Sample Coverage Matrix [4]*

# Test Program

We found that generating coverage matrices for programs that solely consist of loops is not effective for evaluation because if a statement is covered at least once, it is considered covered. In that case, all statements are covered, and each statement is equally suspicious. That indicates that we need to find a program with statements which are never covered in any loop iteration in some test cases. This could be a disadvantage of these fault localisation tools because they are not suitable for all programs. We decided to follow [1]’s example and choose a program with no loops and more branching. We wrote a simple program that provides certain information about a year, e.g. whether it is a leap year, whether it is in the future, etc. The source code is provided below for reference:



Picture 17 Source Code of the Test Program

We found that it is not enough to come up with a suitable program for evaluation. For this particular program, to make sure there are failed cases in the coverage matrix, we needed to find suitable years for bugs to manifest themselves. 3 bugs are intended to be introduced for testing:

|  |  |  |
| --- | --- | --- |
| **Faulty Line #** | **Bug Description** | **Test Value for Bug to Show** |
| 16 | year % 400 == 0  is changed to  year % 400 != 0 | a year that is not divisible by 400 and is not a leap year, e.g. 1900 |
| 27 | <= is set to < | Current year is not the future, so the test value is the current year - 2018 |
| 54 | 1938 is changed to 1939 and 1950 is changed to 1951 | 1938 (World Cup took place in 1938) |

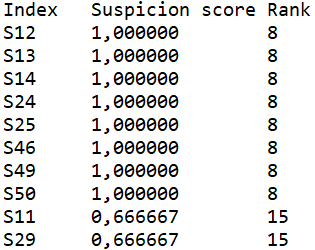
Table 5 Bug Descriptions for Analysis

# Analysis

## Assessment Criteria

We measure the performance of a fault localisation algorithm with the number of statements that need to be discovered before a bug is located. Let Xalgorithm be the score reflecting the performance of a fault localisation algorithm:

where n is the total number of statements and b is the order number of the faulty statement. reflects the percentage statements that need to be checked before the bug. For example, consider the output below:

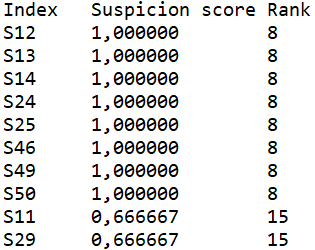
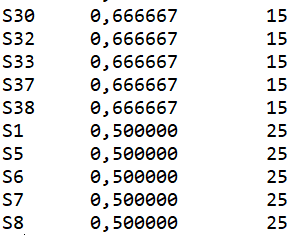
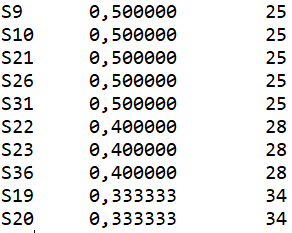


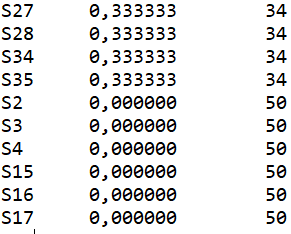
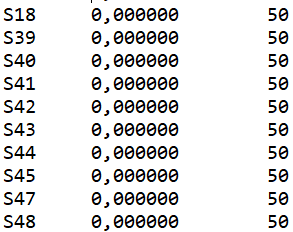
Picture 18 Sample Output for a Fault Localisation Algorithm

Suppose the bug is at line 13, so b = 2 because statement 13 is in the second place. If n = 50, then  2%. That means that the lower is, the better the algorithm performs.

## 1 Bug

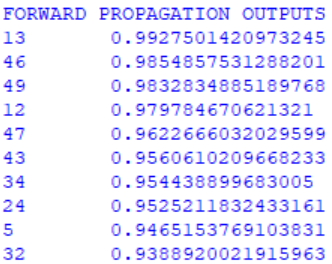
[1] contends Tarantula is mostly useful for programs with a single bug, so we decided to test Tarantula and BPNN for a single bug first. 6 test cases were generated randomly, i.e. random years between 1850 and 2050. The coverage matrix excludes statements like closing brackets }, comments or blank lines, so the statement numbers are not the same with those in Picture 17. Overall, there are 50 lines, and the buggy statement is #13, which is line 16 in the source code (see Picture 17). Picture 18 shows the output for Tarantula.

Picture 19 Outputs for Tarantula

It is evident that Tarantula is close, only 1 statement away from the bug, and XTarantula = 1/50 = 2%. BPNN performed better. Here are the first 10 suspicion scores generated by BPNN:



Picture 20 BPNN Output for Test Program

The error at the last 6000th iteration is 0.000. BPNN is 0 statements away from the bug, so XBPNN = 0%.

For a more logical proof, the position of the single bug is changed to a different place to see the performance of these two debuggers.

In the below experiments, 100 test cases are used to ensure the credulity of the trial.

|  |  |  |
| --- | --- | --- |
| **Position** | **XTarantula** | **XBPNN** |
| #40 | 0% | 10% |

Table 4 Comparison of Tarantula and BPNN

Overall, BPNN performs worse than Tarantula in the above cases.

## More Than 1 Bug

As our program is short, the cases of two and three bugs are considered. In the two-bug case, the bugs are located at statement 13 and 22. The X score is denoted by the number of statements has to be examined before finding both of the bugs.

XTarantula = 64% and XBPNN = 56%

In the case of three bugs, the bugs are located at statement line 13, 22 and 40, and the result shows that the XTarantula = 82% and XBPNN = 56%.

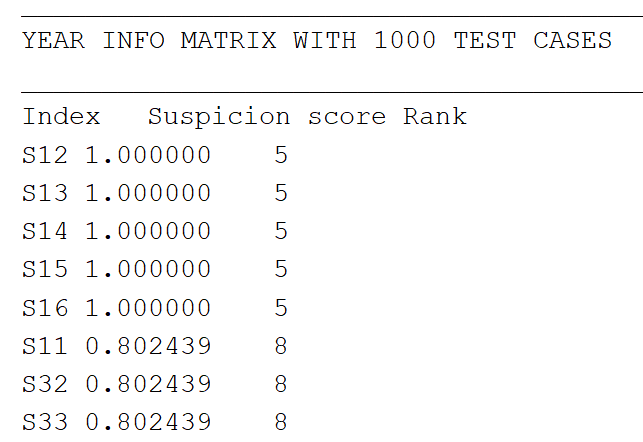
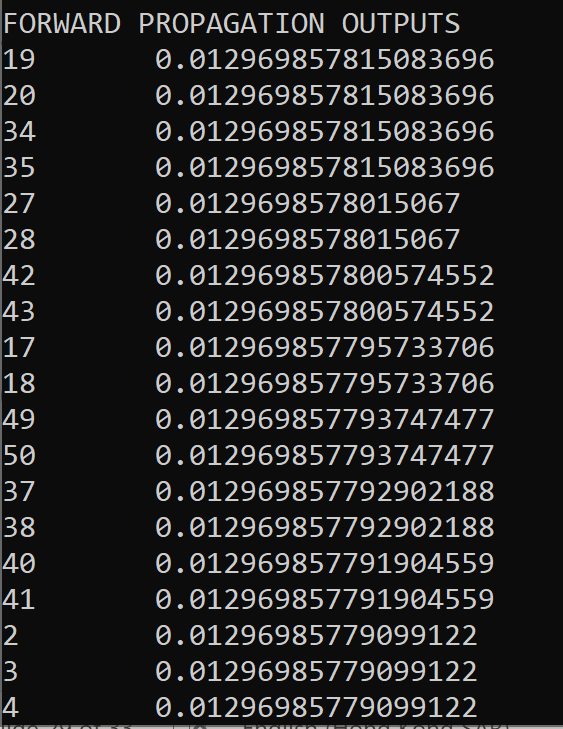
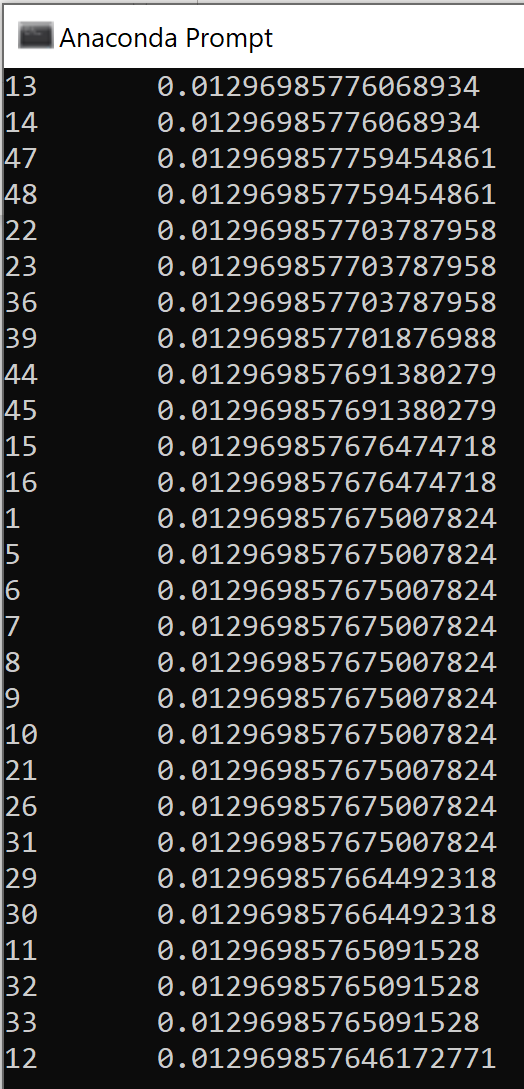
The conclusion is that BPNN performs better than Tarantula in the cases of multiple bugs.

## Increasing the Number of Test Cases

As well as comparing the debugging tools by different number of bugs, we also did the experiments to see whether the number of cases will affect the performance of these two debugging tools. In this part, we use 1000 test cases and the bug is located at statement 13.

XTarantula = 2% and XBPNN = 44%.

When increasing no of test cases, Tarantula performs significantly better than BPNN.

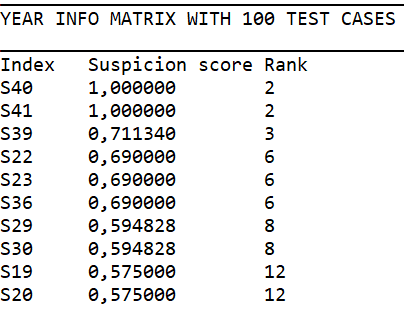
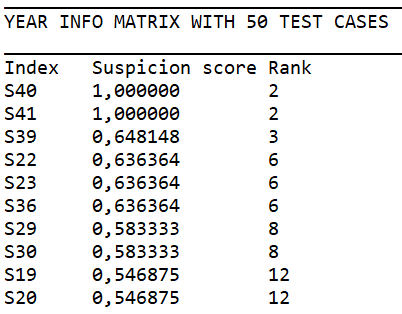
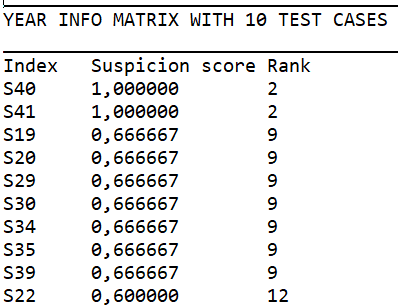


Picture 21 Tarantula and BPNN Output

## Adjusting the Number of Failed Test Cases

We would like to find out whether the ratio of failed to successful test cases affects the accuracy of the algorithms’ predictions. For example, does reducing the number of failed test cases mean algorithms do not have enough information to identify the bug? Or does increasing the number of failed test cases introduce noise to the test data and confuse the algorithms? A bug is injected at line 40 as follows:

It must be noted that since years are generated randomly, we had no control over the number of failed test cases, so we introduced random years that are guaranteed to produce a failed test case deliberately into the data set, according to the ratio of failed test cases. Define RF as the proportion of failed test cases in the coverage matrix, i.e. if RF = 1/3, then 33.3% of test cases are failed. As a sample output, see pictures below for RF = 1/3 and different numbers of test cases.



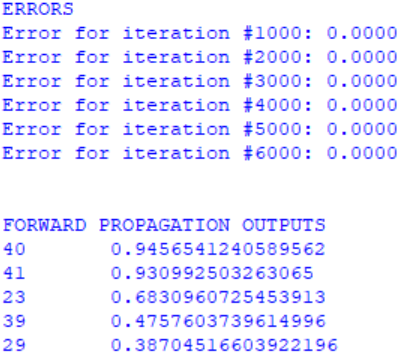
Picture 21 Tarantula's output for Different Numbers of Test Cases with SF = 3 : 1

XTarantula = (1 – 1) / 50 = 0% for all numbers of test cases above. Here is the comparison of BPNN and Tarantula with 10 test cases and RF ranging from 20% to 50%.

Xalgorithm (%)

Diagram 1 Comparison of BPNN and Tarantula with RF = 33.3% and the number of test cases = 10

It should be noted that BPNN performs better with a larger number of test cases. For example, in the graph above, for RF = 1/3, BPNN ranked statement 40 in the 13th place, so XBPNN = (13 – 1) / 50 = 0.24. However, if the number of test cases is increased to 50, BPNN’s ranks statement 40 in the first place (no L2 regularisation) :

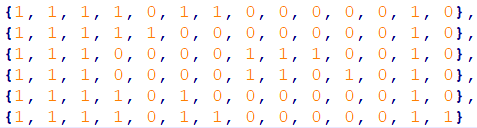


In summary, Tarantula’s output is equally precise with RF values, but BPNN’s precision depends on the number of test cases. This is because it is a learning model, i.e. it is supposed to improve with a larger sample size.

# Appendix

## Matrices Used in Implementation

### Matrix 1 (from [1], correct ordering: (7, 6, 4, 1, 2, 3, 13, 5, 8, 9, 10, 11, 12))



### Matrix 2 (from [4], correct ordering: (8, 1, 5, 10, 9, 6, 3, 7, 4, 2))



### Matrix 3 (from [2], correct ordering unknown)



# References

[1-7]

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," presented at the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA, 2005.

[2] W. E. Wong and Y. U. Qi, "BP Neural Network-based Effective Fault Localization," International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 04, pp. 573-597, 2009/06/01 2009.

[3] J. McCaffrey. (2017). Neural Network L2 Regularization Using Python. Available: <https://visualstudiomagazine.com/articles/2017/09/01/neural-network-l2.aspx>

[4] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, A Crosstab-based Statistical Method for Effective Fault Localization. 2008, pp. 42-51.

[5] M. Mazur. (2015). A Step by Step Backpropagation Example. Available: <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

[6] J. McCaffrey. (2017) Neural Network Back-Propagation Using Python. Visual Studio Magazine. Available: <https://visualstudiomagazine.com/Articles/2017/06/01/Back-Propagation.aspx?Page=1>

[7] ‘Chi-squared distribution’. Wikipedia. Available: <https://en.wikipedia.org/wiki/Chi-squared_distribution#/>. Accessed: 28 November 2018.