Recovering and Tracing Links between Software Codes and Test Codes of the Open Source Projects

Amir Hossein Rasekh
Computer Science and Engineering Department
Shiraz University
Shiraz, Iran

Amir Hossein Arshia
Computer Science and Engineering Department
Shiraz University
Shiraz, Iran

Seyed Mostafa Fakhrahmad*
Computer Science and Engineering Department
Shiraz University
Shiraz, Iran

Mohammad Hadi Sadreddini
Computer Science and Engineering Department
Shiraz University
Shiraz, Iran

Abstract—**One of the most important controversial issues in the design and implementation of software is the functionality of the designed system. With impressive efforts of different software teams in the field of the system, the primary concern of the developers is its proper and error free functioning of the whole system. Therefore, various tests are defined and designed to help software teams to produce error free software or software with minimum error rate. It is difficult but important to find a proper link between written test class and the class under the test. Discovering these links is useful for programmers to perform the Regression Test more efficiently. In this paper, we are trying to propose a model for the recovery of traceable links between test classes and the classes under the test. The presented model comprises four sections. Firstly, we retrieve the name of similar classes between the test class and source class. Afterward, we extract the complexity, Cyclomatic and design metrics from the source codes and the test classes. Finally, after creating a train set, we implement the data mining algorithms to find the potential relationship between unit tests and the classes under the test. One of the advantages of this method is its language independence; furthermore, the preliminary results show that the proposed method has a good performance.**

Keywords—Unit Testing; Source Code; Similarity; Software Engineering; Open Source; Data Mining

I. Introduction

Designing software without observing software engineering principles is like building a house without a standard and engineered plan. Unfortunately, many producing software companies do not follow the principles of engineering or remove some of the stages, especially the test phase of the software development cycle. This reduces the cost of production of software but multiplies the cost of its support and maintenance. This multiplication of the cost of support and maintenance happens due to most of the problems of the program are resolved in the test phase.

Software test is the software evaluation process to ensure the proper functioning of the various events. In other words, software test is finding the possible errors of software during its use to have software which performs correctly, properly and optimally. The more software could work with different events the better the application performs. A good test refers to the test that in which there is more probability of finding undetected errors by the evaluation process. A successful test is a test which can find at least one undetected error. The test just shows the existence of error not the non-existence of error. Finding no error in the test doesn't mean that it is not an error-free program. The software testability criteria are as follows:

- Operability: The software can be better assessed when it operates in more environments.
- Observability: having the capacity of observing the results of the evaluation.
- Controllability: having the ability to manage the automated tests, such as the capacity to operate the unit tests with JUnit in Java language automatically.
- Decomposability: assessment could be more purposeful.
- Simplicity: reducing the architectural complexity and application logic.
- Stability: for evaluation needs little changes.
- Understandability: having the capacity of understanding the design and correlations between the components.

Coding and testing are two activities that are entirely integrated with the agile method of software development. These two activities require that the programmer frequently swaps between software source code and test codes.

Unfortunately, the links between software codes and test codes are implicit and hidden; therefore, for developers discovering these links to find connections between test cases and corresponding software codes is very time-consuming.

A. The Beginning Time of the Test

During the Software Development Life Cycle that is called SDLC, test starts and it lasts until the establishment of the software. However, all these tests depend on the model of the development that the company would operate. For example, in

the waterfall model the test is run in the software production phase, but in the incremental model the test is repeated at the end of each increase or change. In every stage of the SDLC, the analysis and approvals required for the test are considered. Reviewing of the design in the design phase to the aim of design improvement in the area of test is also considered. After the completion of the test by a developer, the test is classified as a unit test.
Testing during the SDLC has the following advantages:

- Reducing the production time
- Reducing the costs
- Reducing the reworking
- Reducing the software errors
- Increasing productivity
- Increasing software quality
- On time delivery to the taskmaster
- Improving the customer satisfaction

B. Unit Test

Unit Test is a technique used to test small units of software source code and also ensures they are working properly. In this technique, the integrity of each piece of software source code, which is called ”unit”, is evaluated using another code written by programmers. In object-oriented languages, this is usually done using a separate class, although it can also be done using only one method. Ideally, each test is independent of the others tests. Unit tests are usually handled by software developers. Method of Unit Test can vary from the evaluation of the result on paper to automatically run multiple tests and analysis of the result by the program itself.

C. The Importance and Benefits of Unit Test

- The compilation of the code does not indicate its correct performance. There need to be methods to test the system. You are not only paid to write code, but you are paid to create executable code.
- In a long time, writing unit tests will result in producing high quality codes. For instance, suppose you've developed a system. Today, an employer asked you to add new functionality to the application. To apply the changes, for example, it is required to modify a portion of the existing code, as well as to add new classes and methods to the program. After the request, you must ensure that the prior parts of the system worked until a few moments ago, now working as before. The volume of written code is high. Manual testing of individual cases may not be possible in terms of time and cost. A unit test is a way to ensure that the delivery of work to taskmaster would happen with no error. In this case, refactoring of the existing code will be done properly, because previous tests can be run immediately; moreover, we can assure that the system performs correctly.
- The procedures of the conducted experiments in the future would become an important reference to understand how different parts of the system perform. How are they calling, how should they be given value, and so on.
- Using the unit tests, we can consider and assess the possible worst-case scenarios before the outbreak.
- Writing unit tests during operation can make the developer break the parts into smaller units that are capable of independent study. For example, suppose you have developed a method that after three different operations on a string provides a specific output.
- These tests are considered as an ideal part of the process of software development because of their automatic execution.

D. Regression Test

The regression test is a method to test the software. The purpose of regression test is to find new software problems or regressions. The purpose of regression test is to assure that new changes such as changes mentioned will not cause a defect or new error in the software. One of the main reasons for doing regression test is to determine if a change in one part of the system can also affect other parts of the system or not. Among the most common technique of performing regression tests is to apply those tests which were done well and successfully before the application of the new changes in the software. Again, after applying new changes, those tests are applied to the software and are examined whether plan behavior changed after applying new changes; moreover, determine if the deficiencies have already been fixed or are not re-emerged and also determine if the already fixed deficiencies have re-emerged or not.

Unfortunately, at present, connections between application codes and test codes are not very rich, and these links are not easily visible. Even when integrated development environments support building test cases based on the generated classes, finding a proper link between the class code and the test is difficult.

Test cases are a valuable source of documentation for developers which they change them continuously to reflect changes in the generated code of their software and maintain an effective regression analysis. Maintaining links between application codes and test codes are an excellent source for selectively testing the software after applying changes in the generated code.

Discovering these links can also be used in the regression analysis. In large-scale software projects, regression test in the form of retest all approach is a time-consuming duty. In particular, operating some of the cases of the test can take hours or even days of time. So developers cannot test software system quickly or even at an acceptable time. Researches show that only 30% of developers, after applying changes in the generated code of their application, they thoroughly tested it again; however, only 70% test those functions that they have changed them.

II. Related Works

There are a variety of methods for discovering the links between software codes and test codes. These methods could then be classified based on the method used. Some of these methods are based on Heuristic Algorithm, some are based on Information Retrieval, some are based on Data Mining and, finally, there are some methods based on Machine Learning.

In recent years, some methods are proposed to retrieve and manage the connections between test classes and software code classes. The researchers proposed the waterfall method to improve the testability of complex classes [1].

In 2007, Bouillon and et al. [2] suggested the Eclipse JUnit to call static graphs. They used a Java icon to identify and communicate the description of generated code of the software.

In 2004, Bruntink and Deursen [3] offered two methods to combine the test case and software generated code. The first method used a name convention and handheld communications for mapping the code functions to the requirements model. In the second method, they proposed the connection between the test case and the code functions with time stamps of the test cases and time-stamp of the code functions.

Ren and et al. [4] proposed a system called Chianati. This system is a software add-on for the Eclipse, which can evaluate the impact of changes in the products with the help of software generated code identification. The system analyzes the changes by the developer between different versions of the system; then it maps them in the test case by analyzing the call graph. Moreover, this system can be implemented after changes; thus, it is not able to identify and maintain the connection between the generated code and test case. This method is also useful to understand the application and test the regression along with effective analysis.

In 2012, Hurdugaci and Zidman [5] wrote a plug-in for visual studio that mapped the changes in the software generated code to the case test code for a change in a piece of code.

In 2009, Rompaey and Demeyer [6] implemented experimentally naming convention, the last call before assertion, Latent Semantic Indexing [8] and co-evolution approach. They discovered the test class naming convention by eliminating the keyword ”test” of classes in the software generated. The last call before assertion recognized those classes under the test by the inspection and reviewing of the method in JUnit. It is a Heuristic approach that makes a distinction between the test classes and helper classes. In summary, the last call before the assertion assumes that the test methods immediately call the actual test classes before the confirmation of the orders. The latent naming convention is an Information Retrieval techniques that identified test classes based on context similarity of the JUnit and code classes of the software. Finally, the Co-evolution approach assumes that a JUnit test class is completed again with a relevant test. Their results have shown that the naming convention is more accurate than other methods. Moreover, the latent naming convention has been successful in identifying the relationship between several types of artifacts such as those functions under the test written in the natural language and program codes. They also showed that the use of The latent naming convention to identify relationships between JUnit test class are not satisfied. Naming convention shows a one by one relationship between the test cases and the software generated code.

In 2010, Qusef and et al. [9] proposed the use of data flow analysis to overcome some restrictions. They considered the test classes as a series of classes that affect the results of the latest announcement of each test. The authors examined decomposability and accessibility based on the dependency of the analysis.

In 2011, Qusef and colleagues [10] proposed using slicing to identify a set of classes that are generated by software and have an effect on commands. It is called SCOTCH. Their results are displayed on three system software called ArgoUML, AgilePlanner, Ant Apache. The advantages of SCOTCH are bright compared to conventional naming techniques and the last call before assertion as well as data flow analysis. In particular, words in class names are important.

In 2014, once again, Qusef and et al. [11], developed their 2011 system based on text filtering strategy from internal and external information communication and called it SCOTCH+. In their work they used name similarity; moreover, they took their results on software systems and with this method they improve their previous result.

III. The Proposed Method

In this paper, a method has been introduced based on data mining algorithms to link between the code classes and the test classes. The proposed method includes the following sections.

1) Retrieving the name of those classes which have the same name in both test classes and source code classes.
2) Extracting features of the source code and test classes:
    a) Extracting complexity metrics
    b) Extracting Cyclomatic metrics
    c) Extracting design metrics
3) Creating the train test and the test set
4) Operating the data mining algorithms on the obtained file.

Figure 1 and Algorithm 1 demonstrate the full implementation of the proposed method and the pseudo codes presented in this paper. These parts will be explained in the next sections respectively.

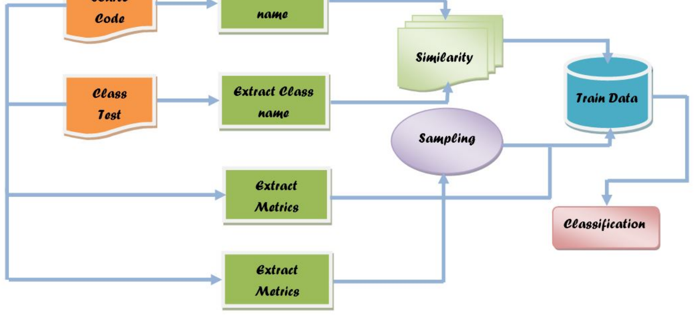

Fig. 1: An overview of the whole system

Algorithm 1 The Pseudocode of the Proposed Method

```
for each (Classname T in TestFolder) do
  for each (Classname C in CodeFolder) do
    if (T equal C) then
      $add \leftarrow T\ to\ NamingConventionList$
    end if
  end for
end for

for each (Item in TestFolder and CodeFolder) do
  $extract\ (Complexity\ Metrics)$
  $extract\ (Cyclomatic\ Metrics)$
  $extract\ (Design\ Metrics)$
end for

for each (Item in TestFolder) do
  if (Item in NamingConventionList) then
    $add \leftarrow Item\ and\ its\ features\ to\ TrainSet$
  end if
end for
```

TABLE I: Comparing Similarity Between Two Terms

Term1	Term2	Jaccard Similarity	Proposed Similarity
PropertiesEcho	PropertiesEcho	100%	100%
PropertiesEcho	EchoProperties	100%	85%

A. Recovery of the Name of the Same Classes Between the Test Class and Source Code Class

One of the common approaches among the open source community is the approach of naming the set of test classes. The naming convention of the test class is for selecting the name of the test class. The keyword *'test'* has been located before or after the test name. For example, a class called *TestSample* or *SampleTest* is implementing a test set associated with a class called *Sample*. However, unfortunately, the developers do not follow the class definition of this approach when naming their test series. Accordingly, to make connections between test cases and source codes, initially, we should act in accordance with similarity criteria.

To find such a relationship, initially, the class brand names along with the packages used in this class from the test files should be derived. In particular, the terms used in the class names have much more importance than the name of the other terms in the content of the class. However, it is believed that the terms outlined in the class name offer primary key information, especially when the naming conventions are applied. The names of source code should be scrutinized by using Jaccard similarity criteria with the threshold of 100 and make links between those classes which have quite similar names.

In this method, to find the similarities between the two words, based on the alphabetical order, this task was done which a number of similar letters in the two words divided by the total number of words were counted as it shown in Table III Algorithm 2.

naming conventions create a one-to-one relationship between the test class and the code class based on names, so when there is more than one class tested or when the developers are not able to follow the naming based on the smart technology, a shortfall has happened.

Algorithm 2 The Pseudocode of Similarity Measure

```
for each item A in list do
  if (A matches B) then
    return y
    for each char in A or B do
      if (char in both A and B) then
        $Di \leftarrow 1$
      else
        $Di \leftarrow 0$
      end if
    end for
  end if
end for

$similarity \leftarrow sum(D)/(|A| + |B|) * 100$

return similarity
```

B. Feature Extraction of the Source Code and Test Classes

Using the model referred to in phase A, only make links between test cases and source codes for some of the test cases should be found out. If there is no possibility to find such links by class name, the feature extraction model should be used.

In this part, the features such as the technical metrics of the software have been extracted from both sides of the source code and the test classes. These technical metrics are quantitative criteria that can be used for product evaluation. In this paper, complexity metrics, cyclomatic metrics and design metrics have been used.

1) Complexity metrics: In this paper, Halstead complexity measure which was proposed by Howard Halstead in 1977 was used to determine the complexity metrics [12]. These metrics identify those software criteria that reflect implementation or express the algorithm in different languages; also, these metrics are independent of the operation on a specific platform.

Halstead aims to identify the properties of software and relationship between them. For these metrics, a series of initial scale should be applied when the design is complete. These scales are introduced as follows:

n1: number of specified operators that appear in the program
n2: number of specified operands that are displayed in the program
N1: total number of operators
N2: total number of operands
The properties extracted from the Source Code and test file are shown as follows:

- The number of program lines: in this feature, the number of lines of each class of the program was derived.
- The length of the program: the length of the program is the total number of operators and operands.

$$ProgramLength : N = N_1 + N_2 \quad (1)$$

- The number of words of the program: the number of words of the program is the total number of unique operators and operands.

$$n = n_1 + n_2 \tag{2}$$

- Estimation of the length of the program: the following formula can be used for estimation of the length of the program.

$$N = n_1 log_2 n_1 + n_2 log_2 n_2 \tag{3}$$

- Volume: volume is the contents of information of the program. Volume size describes the implementation of an algorithm. V calculation is based on the number of operators and operands executable by the control algorithm. Therefore, V sensitivity to LOC criteria is less than code.

$$V = N * log_2 n \tag{4}$$

- Hard/ Difficulty: the difficulty of the program or error-proneness is assigned by the number of unique operators in the program. The difficulty is also proportional to the quotient between the total number of operands and the number of individual operands. For example, if the same operands are used many times in the program, it is probably more prone to errors.

$$D = n_1/2 * N_2/n_2 \tag{5}$$

- Program effort: program effort (implementation) or understanding of a program is proportional to the volume and level of difficulty of a program.

$$E = D * V \tag{6}$$

- Program level: program level (L) is the inverse of the probability of error in the program. This means that a lower level program is more prone to errors than a higher level program.

$$L = 1/D \tag{7}$$

- Program Time: time program is proportional to the efforts. The experiments can be used to calibrate this quantity (amount). Halstead has shown that the division of effort (E) by 18 is an approximation for the seconds of time.

$$T = E/18 \tag{8}$$

2) Cyclomatic metrics: In this paper, McCabe criteria were used for Cyclomatic metric. Thomas McCabe proposed this metric in 1976 [13] for the measurement of the complexity of the software. He also considered the number of independent paths covering an entire module or method as the complexity of it. There are different methods to calculate the number of independent paths in the method, and the most formal method is plotting the control graph of the flow of that method. After plotting the control flow graph using the following formula, we can calculate the number of independent paths. This technique is used in the classes of the source code, but the test classes use the properties of the number of calling methods.

$$Cyclomatic\ Complexity = EN + 2 \tag{9}$$

$$E : Number\ of\ graph\ edges \tag{10}$$

$$N : Number\ of\ graph\ nodes \tag{11}$$

3) Design metrics: Software design technique can be useful in evaluating software. In 1974, Steven and colleagues, at the beginning of their work, introduced coupling on the structural development of the content as ”measuring the strength of the relationship between a module with another module”. The size of the inter-correlation between two objects is called coupling. For example, object ”a” and object ”b” are coupling if a method of the object ”a” is called by a method of ”b” or by accessible variables of the object ”b”. When the defined methods in a class are called by methods or features of the other class, classes are coupled with each other.

Another metric for designing software is Di. Di is a metric for interior design which has the factors related to the internal structure of the module. Di is a basic design metric to evaluate the cohesion of design. The second design software metric is the De. De is an external design metric. De focuses on an external communication of a module with other modules in a software system. So De is a basic metric designed to evaluate coupling of design. The third and the last is the compound design metrics. Dg value is the sum of Di and De.

Coupling factor will display the decimal number that represents the number of communication between the non-inherited classes.

C. Creating the Train Set

For the classification and creating the train set, label one was allocated to the names of those classes which are exactly matched to the source code class which were obtained in phase A; Additionally, those features in phase B from the source codes and test codes were extracted. The sampling method was applied to add the label 0 to the train data.

To perform this process, to determine the distribution of the files, the remaining files were classified.

Then a sample of each source code and test code with their features with zero labels to train data is added. Finally, the train set with features and labels 0 and 1 is available.

D. The Implementation of Data Mining Algorithms on the Obtained Data File

At this phase, the data mining algorithms on the obtained train data from phase C from the source code and test class is run. In this paper, Bayesian learning algorithms, RBF network, logistic regression, SVM, Decision Tree and Part from the data mining algorithms are applied.

1) Naive Bayes Algorithm: A very practical approach to learning is the learning Bayesian method, which has been able to provide useful practical solutions [14].

The Bayesian reasoning method is based on the probability to draw inferences.

This method relies on this principle that there is a probability distribution for each quantity. Therefore, an optimal decision may be concluded by observing a new data and having inferences about the probability distribution. This method requires prior knowledge of a large number of probable values. When this information is not available, we are forced to estimate it.

To achieve that, the basic information which was gathered previously and assumptions about the probability distribution was used. Calculation of the optimal Bayesian hypothesis is very costly.

2) RBF Network Algorithm: This algorithm implements a network with radial Gaussian bias function. Here, the K-Means algorithm is used for the bias function; moreover, the logistic regression is used for the nominal properties and linear regression for the numerical regression. Activating the basis functions is normalized before entering into the linear model, by the accumulation of a number. The K-mean applied to each category separately to extract K cluster for each class.

3) Logistic Regression Algorithm: Nowadays, in most of the studies, by using several other factors, a particular purpose should be achieved to get an optimal level. In statistics, such works are done, and the results are analyzed with different regression methods. In regression by independent variables, the responses are estimated. This response variable is the main purpose of the research [15]. Logistic regression is a special case of regression which is used when the variable is a two or more alternative option, i.e. there are two or more different modes for response variable.

4) SVM Algorithm: This algorithm is used in areas that their data are not separated linearly, and the data are mapped into a higher-dimensional space so that they can be separated in this new space linearly.

The basis of SVM classifier is linear classification. In linear dividing of data, that line should be selected which has the most safety margin.

This algorithm does not stick to the local maximum; furthermore, this algorithm almost works well for high-dimensional data.

5) Decision Tree Algorithm: Trees in artificial intelligence are used to show various concepts such as sentence structure, equations, etc. this is one of the most famous inductive algorithms that are successfully used in different applications. The decision tree has application in practical issues that can be raised to provide a single answer like the name of a category or class. The decision tree is suitable for those issues that are determined by the output value of yes or no. The reason for naming it as tree decision is that this tree shows the decision-making process for determining the categories of input data.

6) PART Algorithm: PART is a class to generate a list of decisions. PART algorithm is used to identify the knowledge, templates and also different rules [16].

IV. Evaluation And Experimental Data

The works done in this field are based on some famous data sets. From these sets, three of them has the most similarity; therefore; briefly we will talk about them in follows:

ArgoUML: it is an open source tool for UML modeling. It contains 1430 classes and 124,000 lines of code. For this tool, a total of 163 test classes by JUnit has been written for it. These 163 classes contain 12000 code lines.

Apache Ant: it is a Java library and a command-line tool which has the duty of delivering files of the construction project. This library contains 851 classes and 108,000 code lines. JUnit writes it a total of 201 test classes for it. These 201 classes include 17000 code lines.

Dependency Finder: it is a set of tool for evaluating and analyzing compiled code in Java language. The kit also contains 498 classes and 29,000 code lines. A total of 193 classes by JUnit is written for it. These 193 classes provide 20,000 code lines.

TABLE II: Characteristics of the Experimented System

Datasets	Source Class (Number)	Test Class (Number)	Links (Number)
Apache Ant	871	75	77
ArgoUML	1424	75	80
Dependency Finder	336	120	96

The results were evaluated by two metrics. Typically, two metric tools called Precision and Recall are used.

The *Recall* is the percent of the correct links found by the proposed algorithm to measure. *Precision* is the accuracy which measures the retrieved candidate list of links.

In this paper, we used the set of test or those samples which are unseen data. The files in phase A cannot be detected. The test file is a subset of data that evaluates the probability of performance of a future model.

Table III shows the number of distinguishable files from phase A on the entire data set separately.

TABLE III: Number of Detected and Undetected Links using Naming Convention Technique

Datasets	Detected	Undetected
Apache Ant	65	12
ArgoUML	62	18
Dependency Finder	67	29

The table IV shows the values obtained from the proposed method of this paper using the Logistic Regression on three data sets; moreover, it shows the results obtained from it using the recall and precision metrics.

TABLE IV: Accuracy of the Proposed System on 3 Open Source Datasets

Code Class To Test Class		
Algorithm	Recall	Precision
Apache ANT	0.94	0.94
ArgoUML	0.88	0.92
Dependency Finder	0.84	0.98

In figure 2, the top line graph shows the results obtained from the Logistic Regression algorithm on all three introduced data set. The first point for each data set indicates Recall, the second point represents Precision and the third point of each data set represents the F-measure.

Figure 3 using the bar graph shows the superiority of the proposed method on the datasets of Apache ANT, ArgoUML and Dependency Finder by SCOTCH+ proposed by Qusef in 2014.

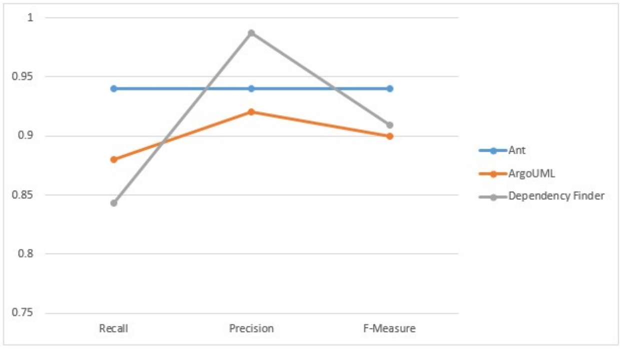

Fig. 2: Accuracy metrics of experimented system

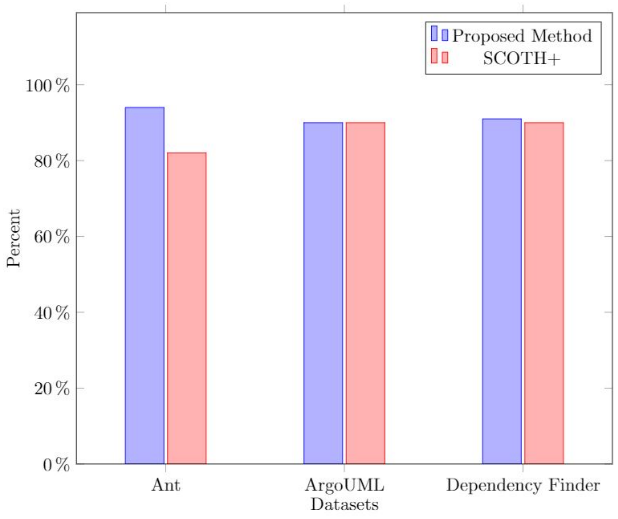

Fig. 3: Comparison proposed method with SCOTCH+ method

V. Conclusion and Future Work

Recently, several techniques have been provided for identifying the links between test classes and classes under the test. In this paper, we try to not only investigate the available algorithm and guidelines but also propose a method to discover the hidden links and track the correlation between software codes by the test classes. The aim of this paper is to propose a method based on the similarity of the names and packages of a class and also feature extraction from the source code and test classes as well as using data mining techniques to discover hidden relationships between software codes class and test classes. Finally, we compared the proposed method with the available methods. The results show the acceptable performance of the proposed method. Discovering these relationships will help programmers to perform the Regression Test more efficiently.

For future works, it could be possible to use feature selection algorithms to extract features with higher impact on the classes. Afterward, it is recommended to weight the features to reach the new results and compare with the previous results.

References

[1] H. M. Sneed, *Reverse engineering of test cases for selective regression testing*, in Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings, 2004, pp. 6974.

[2] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, *EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors*, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, E. Damiani, M. Scotto, and G. Succi, Eds. Springer Berlin Heidelberg, 2007, pp. 101104.

[3] M. Bruntink and A. van Deursen, *Predicting class testability using object-oriented metrics*, in Fourth IEEE International Workshop on Source Code Analysis and Manipulation, 2004, 2004, pp. 136145.

[4] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, *Chianti: A Tool for Change Impact Analysis of Java Programs*, in Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, 2004, pp. 432448.

[5] V. Hurdugaci and A. Zaidman, *Aiding Software Developers to Maintain Developer Tests*, in 2012 16th European Conference on Software Maintenance and Reengineering (CSMR), 2012, pp. 1120.

[6] B. Van Rompaey and S. Demeyer, *Establishing Traceability Links between Unit Test Cases and Units under Test*, in 13th European Conference on Software Maintenance and Reengineering, 2009. CSMR 09, 2009, pp. 209218.

[7] S. M. Fakhrahmad, A. R. Rezapour, M. Zolghadri Jahromi, and M. H. Sadreddini, *A novel approach to machine translation: A proposed language-independent system based on deductive schemes*, Iranian Journal of Science and Technology. Transactions of Electrical Engineering, vol. 38, no. E1, p. 59, 2014.

[8] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, *Indexing by latent semantic analysis*, J. Am. Soc. Inf. Sci., vol. 41, no. 6, pp. 391407, Sep. 1990.

[9] *IEEE Xplore Abstract - Recovering traceability links between unit tests and classes under test: An improved method.* [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5609581. [Accessed: 10-Nov-2015].

[10] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, *SCOTCH: Test-to-code traceability using slicing and conceptual coupling*, in 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 6372.

[11] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, *Recovering test-to-code traceability using slicing and textual analysis*, J. Syst. Softw., vol. 88, pp. 147168, 2014.

[12] M. Halstead, Elements of Software Science, Amsterdam: Elsevier North-Holland, 1977.

[13] Thomas J. McCabe, *A complexity measure*, IEEE Transactions on software Engineering, vol. 4, pp. 308-320, 1976.

[14] P. Lucas, *Expert Knowledge and its Role in Learning Bayesian Networks*, Medicine: an Appraisal, Lecture Notes in Artificial Intelligence 2101, pp. 156-166, 2001.

[15] Xiao, Yuping, M. P. Griffin, D. E. Lake and J. R. Moorman, *Nearest-neighbor and logistic regression analyses of clinical and heart rate characteristics in the early diagnosis of neonatal sepsis*, Medical Decision Making, vol. 30, no. 2, pp. 258-266, 2010.

[16] Cao, Yongqiang and J. Wu, *Dynamics of projective adaptive resonance theory model: The foundation of PART algorithm*, Neural Networks, IEEE Transactions on , vol. 15, no. 2, pp. 245-260, 2004.

[17] J. Tahmoresnezhad and S. Hashemi, *A generalized kernel-based random k-samplesets method for transfer learning*, Iranian Journal of Science and Technology. Transactions of Electrical Engineering, vol 39, p. 193-207, 2015.