PREDICTING METAMORPHIC RELATIONS: AN EVALUATION OF PROGRAM REPRESENTATIONS AND MACHINE LEARNING TECHNIQUES

by

Karishma Rahman

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2020

DEDICATION

I dedicate this thesis
to all those beautiful souls
whom we have lost during
the COVID-19 pandemic of 2020.

ACKNOWLEDGEMENTS

I would like to thank several people for their help in finishing this project. First, my advisor Dr. Upulee Kanewala for her advice and guidance over the past few years. I would also like to thank my lab-mates who have worked along side me and encouraged me. Lastly, I would like to thank my family and friends for their support.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF TABLES – CONTINUED

Table Page

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

Figure Page

ABSTRACT

Testing complex scientific applications can often be a complicated and expensive procedure. A *test oracle* is used to verify the behavior of the software under test. However, difficulties due to the implementation of a test oracle make the process of systematically testing scientific applications more challenging. This problem is known as the *oracle problem*. *Metamorphic testing (MT)* is an effective technique to test these applications as it uses *metamorphic relations (MRs)* to determine whether test cases have passed or failed. Metamorphic relations are essential components of metamorphic testing that highly affect its fault detection effectiveness. MRs are usually identified with the help of a domain expert, which is a labor-intensive task. In this work, a previously developed *graph kernel-based machine learning method* is extended by predicting MRs for functions that perform *matrix calculations*. Then, *semi-supervised support vector machine (S3VM)* is used to build the predictive model for the suggested approach. Finally, *call graph (CG)* information of the functions are used to calculate the graph kernels to predict MRs. The overall result shows that *random walk kernel* performs better than the *graphlet kernel*, and semi-supervised learning can be effective with more unlabelled data. Also, the use of call graph representation presents a new avenue of research in predicting MRs for unseen functions.

CHAPTER ONE

INTRODUCTION

In today's modern world, scientific software is widely used and plays a vital role in making critical decisions in numerous diverse fields, including the nuclear industry, medicine, and the military. Moreover, researchers around the world often rely on this software to perform calculations and conduct simulation that leads to valuable scientific discoveries [1]. Thus, scientists who are unfamiliar with software engineering practices, often find themselves involved in the development of such software[1]. Often, in scientific software, matrices represent data, graphs, or mathematical equations [2]. They can be also be used to get a quick and good approximation for complicated calculations in time-sensitive engineering applications [2]. Therefore matrix calculations are very common in scientific applications. Moreover, matrix multiplication is used in graphics, digital videos, and solving linear equations of particular variables in different applications [2].

Testing scientific applications is hard due to the difficulties associated with defining suitable test oracles [3]. A *test oracle* is a mechanism that determines whether a software produces the correct output for a given test case or not. The test oracle can either be automated or manual [4], and for both cases, the actual output is compared to the expected output. Software testing can be faster, cheaper, and more reliable if testing is done automatically. But for automatic testing, automated test oracles are required, which may not exist or be challenging to implement for scientific software[4]. As a result, many scientific applications fall into the category of 'non-testable programs' [3], where test oracle is not available. Therefore, new methods are

needed for generating test oracles for these programs.

Metamorphic Testing (MT) can be used to alleviate the test oracle problem [5–7], through conducting testing by checking whether the programs behave according to a set of *metamorphic relations (MRs)* [8]. A metamorphic relation specifies how the output should change according to a change made to the input [8]. For conducting MT, first, a set of test inputs known as source test cases are generated. Then, follow-up test cases are generated based on the input modifications specified in an MR. This source and follow-up test cases are executed on the program, and the produced outputs are checked against the output relationship specified in the MR[8]. An MR is violated if the change in output differs from the expected change specified by the MR [8]. Though the violation of an MR is enough to indicate the evidence of faults, only satisfying the MR does not ensure the correctness of the program under test [8]. Identifying a set of MRs that can be applied for testing a program is a crucial primary task when applying MT. Usually, a tester or a developer identifies MRs using their knowledge of the program under test. This manual identification can miss some of the vital MRs that might reveal faults [4].

In 2013 Kanewala et al. [9] first tried to automatically identify metamorphic relations using a feature extraction method where they attempted to predict MRs of programs with numerical inputs and outputs that are represented with simple one-dimensional data structures such as arrays or lists. Further, in 2015 [4], Kanewala et al. developed an approach where they used *graph kernel-based machine learning methods* for predicting MRs.

In this work, the graph kernel-based machine learning method is extended for more complicated programs and is evaluated using other program representations and machine learning techniques. The following lists the contributions that were made in this study.

1. Chapter 3 predicts MRs for functions performing matrix calculations using the above method. Typically, matrices are represented with two-dimensional data structures; thus, the source code of these programs is dissimilar to the ones used in previous work.

2. Chapter 4 uses a semi-supervised classification model to build the predictive model for the graph kernel-based approach. Also comparing the performance of the supervised and semi-supervised models.

3. Chapter 5 evaluates the significance of incorporating call graph information of the functions to calculate the graph kernels to predict MRs.

The rest of this document is organized as follows. Chapter 2 discusses the relevant background information, including an overview of software testing, test oracle problem, metamorphic testing, metamorphic relations, machine learning algorithms. Also, highlight the most interesting recent work in the field of metamorphic relation creation. Chapter 3 presents the experiment of the graph kernel-based methods using support vector machines for a function that performs matrix calculation. Chapter 4 discusses the experiment of the graph kernel-based methods, including semi-supervised support vector machines. Chapter 5 shows the experiment of the graph kernel-based machine learning method by incorporation the call graphs. Chapter 5 presents the results and discussions. Finally, Chapter 6 concludes the paper by discussing the overall results, threats to validity, and future works.

CHAPTER TWO

BACKGROUND

This chapter introduces the background of software testing, describes the test oracle problem, and explains how metamorphic testing works by operating according to the corresponding metamorphic relations. It also illustrates the challenges within MT and the motivation of using machine learning techniques. Lastly, this chapter concludes by mentioning related research works of different techniques for identifying MRs.

2.1 Software Testing

Software testing is a quality assurance activity the system under is executed with test inputs in order to reveal failure[10]. A failure is identified when the produced output is different from the expected one, according to its specifications [11]. Testing requires test input values in order to run on the system that is under test [11], and there should be a mechanism to determine the expected outputs of the system under test for those corresponding inputs to evaluate the results of the test cases.

Ammann and Offutt [11] state that testing could only reveal the presence of failures. Despite how thoroughly it is planned or carried out, there will be limitations of testing activities. The system has to be executed on all likely inputs in all possible scenarios to ensure that it will not fail in the future. Implementing this kind of extensive testing is usually impossible or impractical, due to the enormous size of the input domain [12]. Hence, testers traditionally come up with some test evaluation measures such as test coverage, which helps them to decide whether the system has been tested adequately or not.

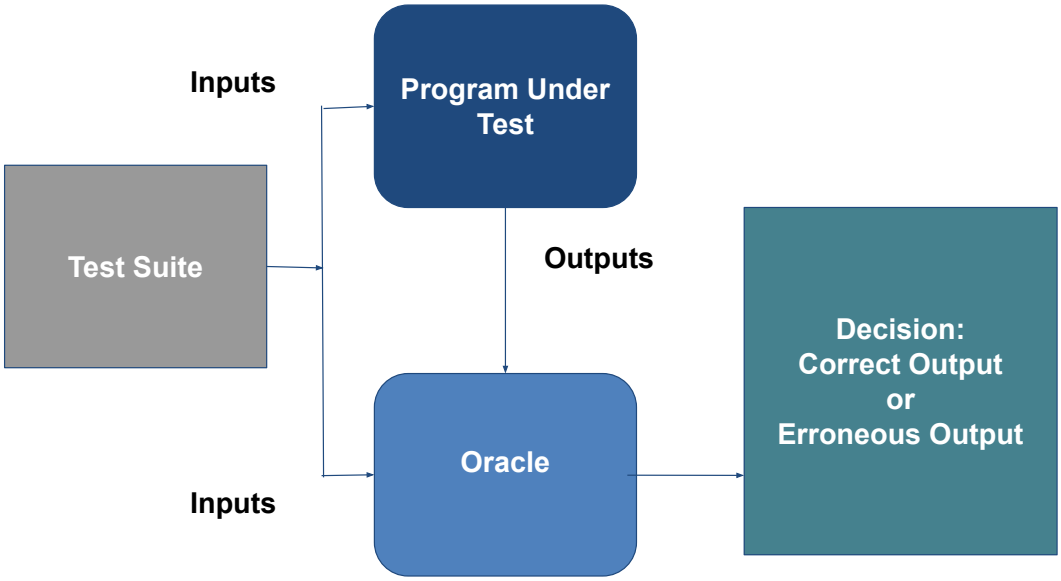

Figure 2.1: Overview of the mechanism of the test oracle for a program under test

2.2 Test Oracle Problem

A test oracle is needed to automate software testing and make the testing process more cost-effective and reliable. A test oracle is a mechanism for determining whether a test case has passed or failed [13]. William E. Howden [14] first introduced test oracles. Most of the time, testers play the role of an oracle, but sometimes an oracle can also be a specification or even another program. Figure 2.1 shows an overview of the mechanism of a test oracle.

One of the most significant challenges in software testing is the oracle problem [13]. The oracle problem or test oracle problem is the difficulty of distinguishing the correct output for a given input [13]. Due to the complexity of scientific software, oracle problem is especially prevalent in these. Thus, often correctness of the outputs is determined manually, which can be error-prone and can lead to missing subtle faults

such as one-off errors. Sanders et al., [15] stated that scientists usually conduct testing in an unsystematic manner due to the lack of background knowledge in software engineering. As a result, techniques are needed to test software automatically without a test oracle to make the process easier.

2.3 Metamorphic Testing

The test oracle problem can be alleviated using metamorphic testing, which is a software testing technique introduced by Chen et al. [5]. It is a property-based testing technique that operates by checking whether the program under test satisfies some previously identified properties [5]. These properties are called Metamorphic Relations (MRs) [5]. The metamorphic relations determine how a specific change in the input should change the output of a program [5]. When the inputs and corresponding outputs of the program under test violate these expected relations, it indicates that some faults exist in the program [5]. Generally, metamorphic testing can be implemented through the following steps (Figure 2.2) [5] :

1. Identify a suitable set of MRs that satisfy the program under test.
2. Create a set of initial test cases.
3. Apply the input transformations specified by the identified MRs in Step 1 to create follow-up test cases for each of the initial test case.
4. Execute the initial and follow-up test case(s) and check if the change in output behaves according to the change predicted by the MR. If a run-time violation of an MR occurs, it can mean that the program under test contains fault(s).

The objective of metamorphic testing is to use newly generated follow-up test cases according to metamorphic relations to identify faults in programs without test

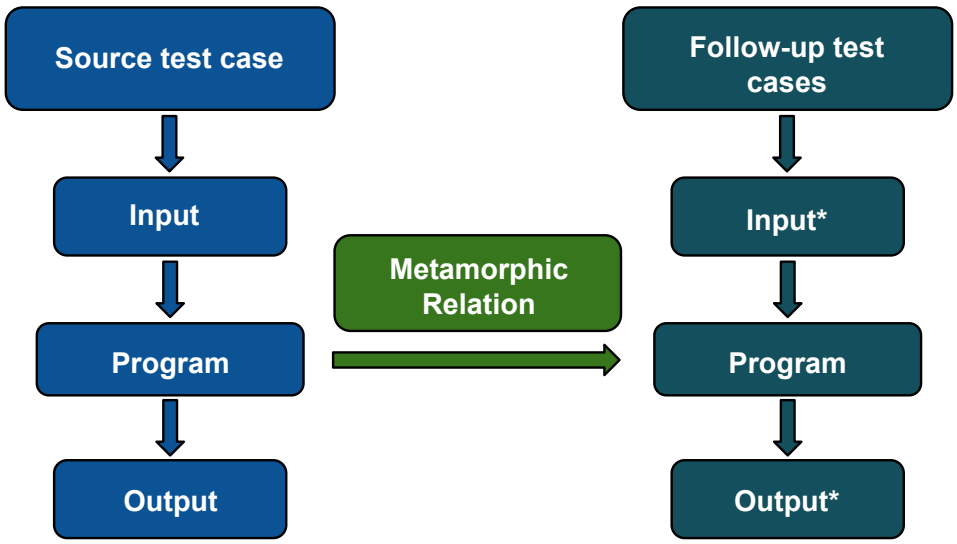

Figure 2.2: The overview of the metamorphic testing process

oracles [5]. As metamorphic testing examines the input and output relationship between multiple executions of the program being tested, it can be used when the correct result of each execution is unknown [4].

The most common and easy example of MT is the implementation of a SINE function y = sin(x). Based on the property of the function, for any input with angle x, the output will stay the same if 2π is added with the input angle (e.g. $y = sin(x) = sin(x + 2\pi)$). This property can be used as an MR. When $x' = x + 2\pi$, then $sin(x') = sin(x)$. The function can be tested using the source test case $(x, sin(x))$ and the follow-up test case $(x', sin(x'))$. If the violation of the MR occurs, which means for the input $x' = x + 2\pi$, $sin(x') \neq sin(x)$, then it is sure that this implementation of the function is a failure.

2.4 Metamorphic Relations

As discussed earlier, metamorphic relations specify necessary relationships between the inputs and the corresponding outputs of a program. Suppose A and B are the input of the matrix multiplication function, the initial output will be O. If matrix A is multiplied by a positive integer n and B remains the same, the change in the output will be $O' = (A \times n) \times B = A' \times B = O \times n$. Here, A' is the follow-up input. Consider a simple example of multiplying two matrices,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$A \times B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix} = O$$

A positive integer of 2 is multiplied to the matrix A. After the multiplication of $A' = A \times 2$ and B, the output is O'.

$$A' = A \times 2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times 2 = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$A' \times B = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 14 & 20 \\ 30 & 44 \end{bmatrix} = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix} \times 2 = O \times 2 = O'$$

Therefore, using this relation of multiplying a positive constant, a follow-up test case can be created for every initial test case, and the output of the follow-up test case can also be predicted by satisfying the change in the output.

In the previous work of Murphy et al. [16], he introduced six fundamental metamorphic relations that can be applied to generate follow-up test cases for

mathematical functions that take an array as input. They are Permutative, Additive, Multiplicative, Invertive, Inclusive, and Exclusive [16]. If the change in the output of a program corresponds to the expectation after reconstructing the input, the function can be deemed to satisfy the corresponding MR. It is essential to correctly identify the metamorphic relations to apply metamorphic testing to the program under test. However, this is challenging for a tester to achieve such a set of MRs, as they do not have enough prior domain knowledge of the programs. Moreover, identifying metamorphic relations often needs to be done manually; as a result, it is a labor-intensive and time-consuming task. So, an efficient and systematic machine learning approach is introduced in this paper to deal with this problem.

2.5 Machine Learning

Recently, machine learning (ML) is being applied to automate numerous software engineering activities, including software testing. ML is primarily a set of algorithms that are used to design models and understand data [17]. ML algorithms are thus stated as data-driven methods by Mohri et al. [18]. Moreover, it combines the fundamental theories of computer science with the concepts from statistics, probability, and optimization [17]. According to Shalev-Shwartz and Ben-David [19], ML is focused on learning by computers; hence, algorithmic ideas are essential. The practice of ML to tackle software-testing problems is comparatively a new domain of research. Various numbers of researchers in this area have published their work in the past two decades [6]. In these studies, different ML algorithms are explored and used to automate software testing. Therefore, it can be used to automate the identification of MRs using prediction analysis for the program under test.

For an ML classification algorithm, when an adequate amount of data is being passed to them, it will build a model and make decision-based on the model. If more

data are added to the model, the machine recognizes the patterns more accurately. Hence, the machine learning method teaches computer programs to make better decisions based on experience [17]. The set of examples used by a machine learning algorithm is divided into two groups. They are called a *training set* and a *test set* [17]. The training set is used to create a predictive model using ML algorithms, and the test set is used to assess the performance of the predictive model [17]. Two types of machine learning methods are investigated in this paper for the automation of the MR identification process.

2.5.1 Supervised Learning

Supervised learning is a machine learning method that uses a set of examples to learn a pattern [17]. The pattern is then used to map the unseen inputs to the desired set of outputs. More precisely, a supervised learning algorithm examines the training data and provides a targeted function, which is used to determine the labels for the unseen test data [17].

2.5.2 Semi-supervised Learning

Because of the abundance of data on the internet today, and most of them being unlabeled, Semi-supervised models are picking up importance as they mostly use unlabeled data to create the predictive model [19]. Moreover, identifying the labels for the training data can be an error-prone activity [19]. Therefore, the addition of more unlabeled data benefits in producing higher classification accuracy scores than using only labeled data [19].

2.6 Related Work

Several previous studies have looked into automatically generating/predicting MRs. Liu *et al.* introduced a new method called *Composition of Metamorphic Relation (CMR)*, where the generation of new metamorphic relations is done by combining existing metamorphic relations [20]. Dong *et. al* conducted a similar study, where *Compositional MR* was generated based on the speculative law of proposition logic [21].

Zhang *et al.* suggested a technique, where an algorithm searches for metamorphic relations in the form of linear or quadratic equations [22]. Su *et al.* also suggested a new method called *KABU*, which can be used to find more likely metamorphic relations by dynamically inferring the properties of the status of a method [23].

Chen *et al.* introduced an approach called *DESSERT*, where DividE-and-conquer methodology was used to identify the categorieS, choiceS, and choicE Relations for Test case generation [24]. Later, Chen *et al.* proposed a tool called *METRIC*, where metamorphic relations were identified with category-choice framework [25].

Kanewala *et. al* [4, 9] showed that, in previously unseen programs, MRs could be predicted using a machine learning method. Features were extracted from CFGs of the functions, and they were then used to create a predictive model [9]. Kanewala et al. [4] introduced *MRpred*, a method that uses a graph kernel-based machine learning approach to predict metamorphic relations for programs that perform numerical calculations. The initial step of this approach is to transform a function into its graph representation, modeling the control flow and the data dependency information of the program [4]. Then they use a graph kernel function to compute a similarity score between two programs represented in the graph representation mentioned above.

The computed graph kernel values are then provided to a support vector machines (SVMs) classification algorithm to create the predictive model, which is used for binary classification [4]. They used a code corpus containing 100 functions that take numerical inputs and produce numerical outputs, to evaluate the effectiveness of their proposed methods [4]. Six MRs are identified; Permutative, Additive, Multiplicative, Invertive, Inclusive, and Exclusive. Their results show that graph kernels improve the prediction accuracy of MRs when compared with explicitly extracted features. Their results also show that control flow information of a program is more effective than data dependency information for predicting MRs, but sometimes, both of them can contribute to increasing the accuracy.

In 2018, Hardin et al. presented a technique to predict metamorphic relations, where they implemented machine learning models using the support vector machines and the label propagation algorithm [26]. Their feature-set contains path information throughout the graph representations of the program under test. Their result shows that label propagation performed better than the SVMs for 5 out of the 6 MRs [26]. Moreover, they conclude that unlabeled data improves the prediction rate of a classifier [26]. These studies were the motivation to use matrix-based programs as the subject programs for the experiments described in this paper.

CHAPTER THREE

PREDICTING METAMORPHIC RELATIONS FOR MATRIX CALCULATION PROGRAMS

3.1 Contribution of Authors and Co-Authors

Extension of manuscript in chapter 3

Author: Karishma Rahman

Co-Author: Upulee Kanewala

3.2 Manuscript Information

[Karishma Rahman and Upulee Kanewala]

MET18: Proceedings of the 3rd International Workshop on Metamorphic Testing

Status of Manuscript:
____ Prepared for submission to a peer-reviewed journal
____ Officially submitted to a peer-reviewed journal
____ Accepted by a peer-reviewed journal
_X_ Published in a peer-reviewed workshop

May 2018, Pages 10–13 https://doi.org/10.1145/3193977.3193983

3.3 Approach

This section describes the extension of graph kernel-based machine learning method for programs that perform matrix calculation.

3.3.1 Method Overview

This approach leverages kernel-based machine learning algorithms to train a classifier and predict the MRs for previously unseen programs. The overview of this approach is shown in Figure 3.1. The method consists of two phases. The first one is the training phase, which starts by generating the graph representation of the functions written by the Java programming language. Each function is then labeled by its corresponding metamorphic relations, which are identified manually for the experiment. Graph kernels are then used to calculate the similarity score between each pair of graphs. The output of the graph kernel function is a similarity score matrix or gram matrix. The final step of the first phase is the training of a classifier using kernel-based machine learning algorithms. The second phase is the testing phase, which extracts the graph representation from a previously unseen function, in a similar manner to the training phase. After that, the similarity scores between the previously unseen graph and each graph for training are computed using the graph kernel function. Finally, the trained classifier is used to predict the MRs for the unseen function.

3.3.2 Function Representation - Control Flow Graph (CFG)

The first step of this method is to convert a function into its Control Flow Graph (CFG). This representation is specifically used since it allows the extraction of information about the sequence of operations performed in a control flow path that is directly related to the MRs satisfied by a given function.

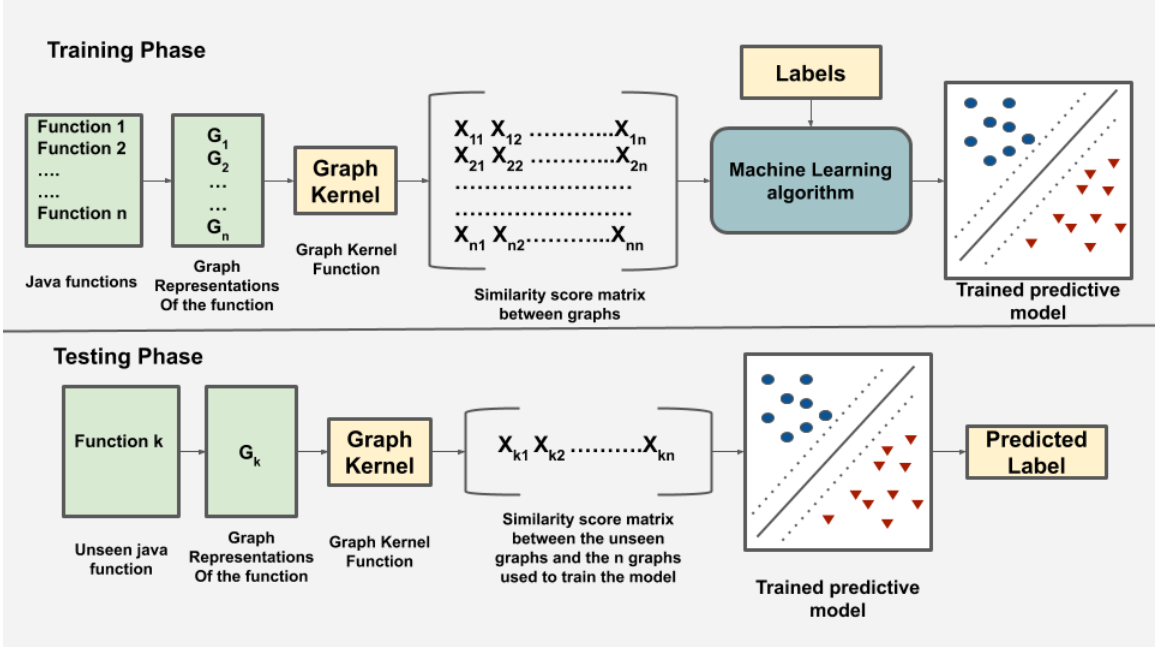

Figure 3.1: The overview of the graph kernel-based machine learning approach

A CFG is a directed graph $G_f = (V, E)$ of a function *f*. Here, x is a statement in *f*, represented by each node $v_x \in V$. The operation performed in each x are labeled $label(v_x)$. Supposedly if x and y are statements of *f*, after execution of x, y is executed. Then it can be said that e is an edge where $e = (v_x, v_y) \in E$. Control flow of f is represented by all the edges, and the starting and the exiting point are represented by nodes v_{start} and v_{exit} respectively [27].

We use the *Soot*[1] framework to create the CFGs. It generates CFGs in *Jimple*: a typed 3-address intermediate representation of the Java code; thus each CFG node represents an atomic operation [28]. We post-processed the generated CFGs by labeling all the nodes indicating the operation performed in each node. In addition, we annotated all the method call nodes in the CFG with their return types. Figure 3.2 represents a function for calculating scalar multiplication of a matrix and its

[1]https://www.sable.mcgill.ca/soot/

post-processed CFG representation.

3.3.3 Graph Kernel-Based Method

As the proposed method leverages kernel-based supervised machine learning algorithms, some descriptions of the kernel method, graph kernel, and machine learning methods are provided.

3.3.3.1 Kernel Method: Kernel methods are a class of machine learning algorithms that can perform pattern analysis and find relations such as classification, clusters, and principal components, etc. [29]. The raw representation of data can be mapped by using kernel methods into a high-dimensional feature space where machine learning algorithms can find the linear relation [29]. Kernel functions allow computing the inner product in the high-dimensional feature space from the raw representation of data, rather than mapping the data into that new feature space and measuring the coordinates [29].

3.3.3.2 Graph Kernel: In different domains such as biology [30], chemistry [31], and social network analysis [32], kernel-based graph comparison and classification method have become very popular. Graphs are the common and natural representation to model such types of data, as they mainly focus on the structure. Gartner et al. introduced graph kernels in 2003 [33]. It is a group of functions that can calculate the similarity score for a pair of graphs by comparing the structure of two graphs. After that, kernel-based machine learning algorithms can use the precomputed similarity scores to perform the classification process. Two types of graph kernel are used for the graph kernel-based machine learning approach [4]. In the previous work by Kanewala et al. [4], a random walk kernel and graphlet kernel are used to predict MRs. The study concludes that the random walk kernel performs better for predicting MRs

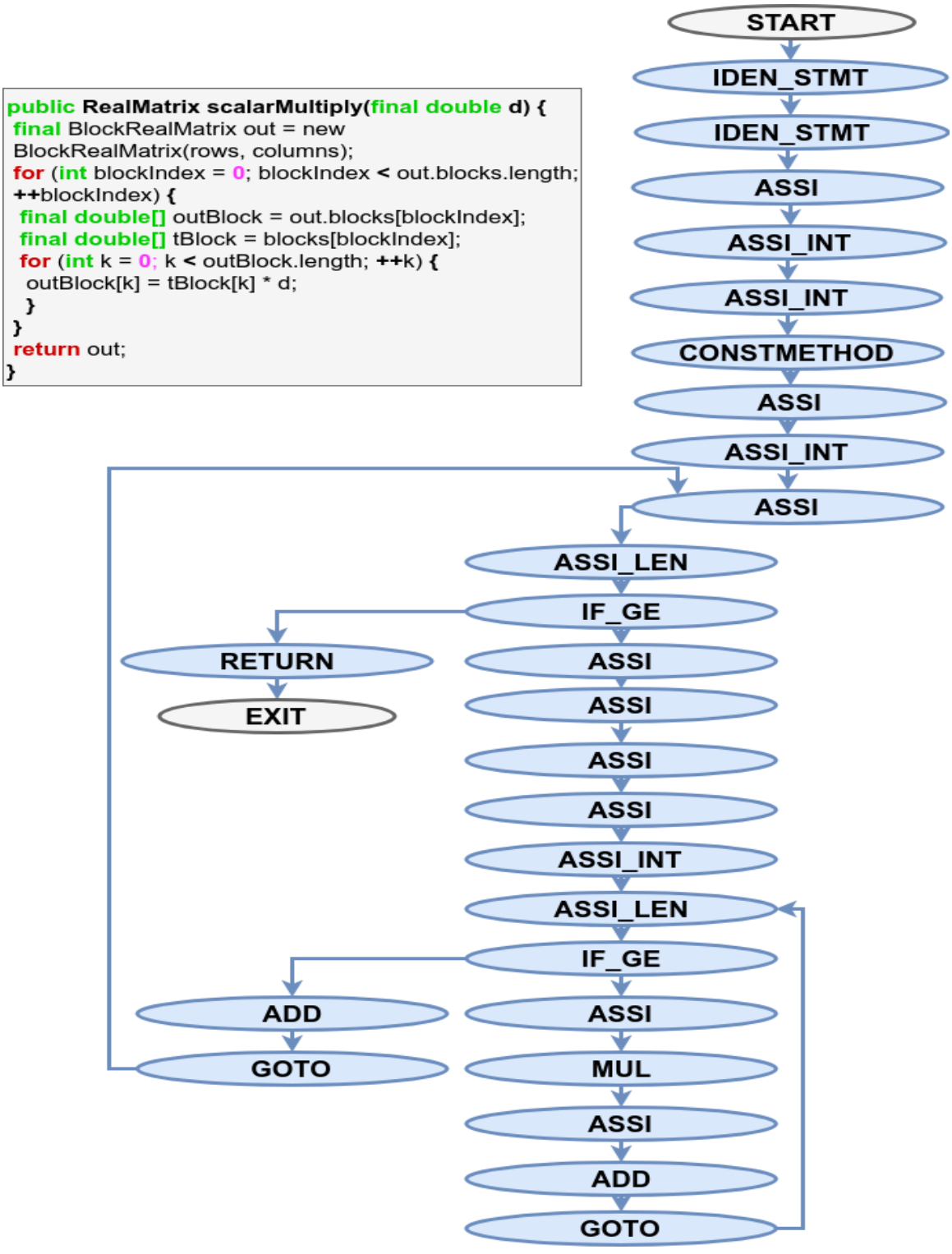

Figure 3.2: Function that performs scalar multiplication and its post-processed control flow graph (CFG) representation

when using CFG representation. In this study, new types of functions are evaluated using both the graph kernels.

3.3.3.3 Random Walk Kernel: After creating the function representation of the programs, the next step is to use a graph kernel to compute the similarity between the graphs. The random walk kernel is based on the idea of counting the number of matching walks in two graphs [4]. It computes the similarity score between two graphs by summing up the similarity scores of all the pairs of walks, which is computed by multiplying the similarity scores of their corresponding step pairs [4]. The similarity score of a pair of steps is originally calculated by multiplying the similarity scores of node and edge pairs that make up the step [4].

The node labels determine the similarity score of a node pair in the following way: if the two node labels are the same, then the pair is assigned a similarity score of one, else different range of similarity score is assigned based on the operation of the nodes. When the operation in two node labels represent two operations with similar properties (but not identical), the pair is assigned a similarity score of 0.75. If the nodes are doing general activity like the assignment of a variable and function calls, it is 0.5. A score of 0.6 is assigned when the nodes represent the condition or logic of any type and also basic operations of elementary arithmetic. If the nodes of the functions represent throw statements, it is 0.3, and finally, a similarity score of 0.1 is assigned for all other cases. Similar to the node labels, edge labels also decide the value assigned for the similarity score of a pair of edges. Here, we only used one type of edge, which shows the flow of control between the operations of a function. So the similarity score for a pair of edges is always one in regard to this study. The definition of the random walk kernel used in this study is described in the previous work of Kanewala et al. [4].

3.3.3.3.1 Definition of the random walk kernel Suppose, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are the representation of control flow graphs of two programs. Let, consider two walks, $walk_1$ and $walk_2$ for G_1 and G_2 respectively. Here, $walk_1 = (v_1^1, v_1^2, \ldots, v_1^{n-1}, v_1^n)$ where $v_1^i \in V_1$ and $1 \leq i \leq n$. Again, $walk_2 = (v_2^1, v_2^2, \ldots, v_2^{n-1}, v_2^n)$ where $v_2^i \in V_2$ and $1 \leq i \leq n$. Edges are represented as, $(v_1^i, v_1^{i+1}) \in E_1$ and $(v_2^i, v_2^{i+1}) \in E_2$. So now we can define the kernel value of two graphs as:

$$k_{rw}(G_1, G_2) = \sum_{walk_1 \in G_1} \sum_{walk_2 \in G_2} k_{walk}(walk_1, walk_2) \tag{3.1}$$

Now the walk kernel k_{walk} can be defined as follows:

$$k_{walk}(walk_1, walk_2) = \prod_{i=1}^{n-1} k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) \tag{3.2}$$

In each step, the kernel will be defined by utilizing the two node pairs and edge pair values as follows:

$$k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = k_{node}(v_1^i, v_2^i) * k_{node}(v_1^{i+1}, v_2^{i+1}) * k_{edge}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) \tag{3.3}$$

Here, k_{node} are the node kernels, which get the similarity score between two nodes. It checks the similarity of the node labels. For the similarity score assignment we also consider grouping the nodes according to its operations. In this study we grouped the mathematical operations using few properties, such as commutative, associative denoted as $group_{com,aso}$, conditional statements denoted as $group_{condition}$, assignment

of variable denoted as $group_{assign}$ and throw statements as $group_{throw}$.

$$k_{node}(v_i, v_j) = \begin{cases} 1, & \text{if } label(v_i) = label(v_j) \\ 0.75, & \text{if } group_{com,aso}(v_i) = group_{com,aso}(v_j) \text{ and } label(v_i) \neq label(v_j) \\ 0.6, & \text{if } group_{condition}(v_i) = group_{condition}(v_j) \text{ and } label(v_i) \neq label(v_j) \\ 0.5, & \text{if } group_{assign}(v_i) = group_{assign}(v_j) \text{ and } label(v_i) \neq label(v_j) \\ 0.3, & \text{if } group_{throw}(v_i) = group_{throw}(v_j) \text{ and } label(v_i) \neq label(v_j) \\ 0.1, & \text{if } group(v_i) \neq group(v_j) \text{ and } label(v_i) \neq label(v_j) \end{cases} \tag{3.4}$$

The edge kernel $k_{edge}$is defined as follow:

$$k_{edge}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = \begin{cases} 1, & \text{if } label(v_1^i, v_1^{i+1}) = label(v_2^i, v_2^{i+1}) \\ 0, & \text{otherwise.} \end{cases} \tag{3.5}$$

The direct product graph approach is used, which is presented by Gärtner et al.[34]. Some modification. is made, which is introduced by Borgwardt et al. [35], for calculating all the walks within two graphs. Here, the direct product graph of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is determined by $G_1 \times G_2$. Following defined are the nodes and edges of the direct product graph:

$$V_X(G_1 \times G_2) = \{(v_1, v_2) \in V_1 \times V_2\} \tag{3.6}$$

$$\begin{aligned} E_X(G_1 \times G_2) = \big\{\big(\big((v_1^1, v_1^2), (v_2^1, v_2^2)\big)\big) \in V^2(G_1 \times G_2) : \big(v_1^1, v_1^2\big) \in E_1 \\ \wedge \big(v_2^1, v_2^2\big) \in E_2 \wedge label\big(v_1^1, v_1^2\big) = label\big(v_2^1, v_2^2\big)\big\} \end{aligned} \tag{3.7}$$

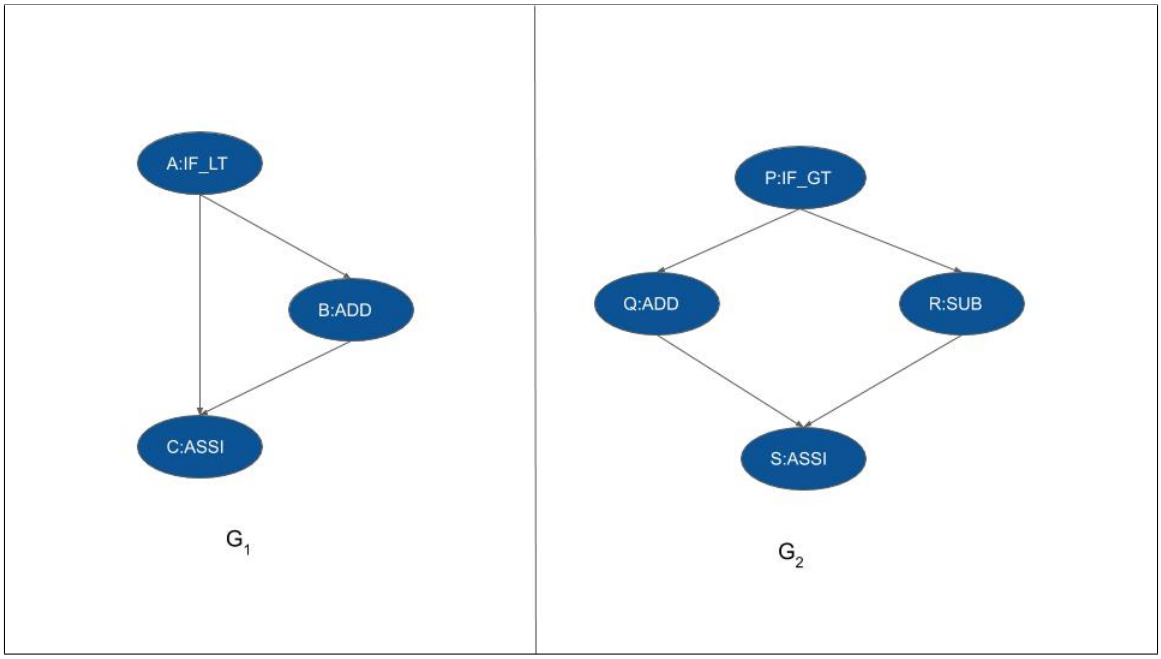

Figure 3.3: Random walk kernel computation for graph G_1 and G_2

Based on the above product graph, the random walk kernel is defined as follows:

$$k_{rw}(G_1, G_2) = \sum_{i,j=1}^{V_X} \left[\sum_{n=0}^{\infty} \lambda^n A_X^n \right]_{ij} \tag{3.8}$$

Here, A_X is denoted as the adjacency matrix of the direct product graph, where $1 > \lambda \geq 0$ is a weighting factor, and n is the path length. The A_X is modified as follows to include k_{step} defined above:

$$[A_X]_{((v_i,w_i),(v_j,w_j))} = \begin{cases} k_{step}((v_i, w_i), (v_j, w_j)), & \text{if } (v_i, w_i), (v_j, w_j) \in E_X \\ 0, & \text{otherwise.} \end{cases} \tag{3.9}$$

For example, in Figure 5.2, G_1 and G_2 are graph representation of two functions, supposing the length of the walks of the graphs are restricted to two walks.

Considering G_1, with walks of length one and two are

$$Length\ 1 : A \rightarrow B, B \rightarrow C, A \rightarrow C$$

$$Length\ 2 : A \rightarrow B \rightarrow C$$

For G_2, with walks of length one and two are

$$Length\ 1 : P \rightarrow Q, P \rightarrow R, Q \rightarrow S, R \rightarrow S$$

$$Length\ 2 : P \rightarrow Q \rightarrow S, P \rightarrow R \rightarrow S$$

Then it computes the similarity score between the two walks of the graphs.

$$k_{\text{walk}}(A \rightarrow B, P \rightarrow Q) = k_{\text{step}}((A, B), (P, Q))$$

$$\vdots$$

$$k_{\text{walk}}(A \rightarrow B \rightarrow C, P \rightarrow R, \rightarrow S) = k_{\text{step}}((A, B), (P, R)) \times k_{\text{step}}((B, C), (R, S))$$

Computation of the similarity between two steps are done as stated below-

$$k_{\text{step}}((A, B), (P, Q)) = k_{\text{node}}(A, P) \times k_{\text{node}}(B, Q) \times k_{\text{edge}}((A, B), (P, Q))$$

$$k_{\text{step}}((A, B), (P, R)) = k_{\text{node}}(A, P) \times k_{\text{node}}(B, R) \times k_{\text{edge}}((A, B), (P, R))$$

$$k_{\text{step}}((B, C), (R, S)) = k_{\text{node}}(B, R) \times k_{\text{node}}(C, S) \times k_{\text{edge}}((B, C), (R, S))$$

Similarity score between two nodes and edges are calculated as follows-

$$k_{\text{node}}(A, P) = 0.6 \text{ (two node labels have same conditions)}$$

$$k_{\text{node}}(B, Q) = 1 \text{ (two node labels are identical)}$$

$$k_{\text{node}}(B, R) = 0.6 \text{ (two node labels have basic arithmetic operations)}$$

$$k_{\text{edge}}((A, B), (P, Q)) = 1 \text{ (both edges have the same label)}$$

$$k_{\text{edge}}((A, B), (P, R)) = 1$$

This concept of the *random walk graph kernel* computation process is used in this study.

3.3.3.4 Graphlet Kernel: Another type of graph kernel method used in this study is the Graphlet Kernel. With a random walk kernel, it is tough to capture major structures like *if conditions*. Because sub-graphs can capture those structures easily, graphlet kernels can be used to calculate the similarity score of a pair of graphs by comparing all sub-graphs of the limited size called graphlets [4]. The similarity score of the pair of graphs is calculated by summing up the similarity score of all the graphlet pairs in those two graphs [4].

The similarity score of a pair of graphlets is calculated as follows: If a pair of graphlets is isomorphic, it calculates the score by multiplying the similarity score between the node and edge pairs in a similar way to the random walk. Otherwise, it assigns a value of 0.1 to the similarity score of those two graphlets. This process is done when the functions are represented as CFGs. Here, only one type of edge is used, which shows the flow of execution between the operations of a function. So the similarity score for a pair of edges is always one. The definition of the graphlet kernel

used in this study is described in the previous work of Kanewala et al. [4].

3.3.3.4.1 Definition of the Graphlet Kernel $G = (V, E)$ is a graph representation where $V = v_1, v_2, \ldots, v_n$ are the n vertices and $E \subseteq V \times V$ is the set of edges. When $G = (V, E)$ and $H = (V_H, E_H)$, H is a sub-graph of G if there is an injective mapping $\alpha : V_H \to V$, such that $(v, w) \in E_H$ if $(\alpha(v), \alpha(w)) \in E$. If H is a sub-graph of G, it is denoted by $H \sqsubseteq G$.

Two graphs $G_1 = (V_1, E_1)$ and $(G_2 = V_2, E_2)$ are isomorphic if a bijective mapping $g : V_1 \to V_2$ such that $(V_i, V_j) \in E_2$ exists. If G_1 and G_2 are isomorphic, it is denoted by $G_1 \simeq G_2$ and g is called isomorphism function.

Let M_1^k and M_2^k be the set sub-graphs of size k for the graphs G_1 and G_2, respectively, and if $S_1 = (V_{s_1}, E_{s_1}) \in M_1^k$ and $S_2 = (V_{s_2}, E_{s_2}) \in M_2$, then the graphlet kernel, $k_{graphlet}(G_1, G_2)$, is computed as

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in M_1^k} \sum_{S_2 \in M_2^k} \delta(S \simeq S_2) \tag{3.10}$$

, where

$$\delta(S_1 \simeq S_2) = \begin{cases} 1, & \text{if } S_1 \simeq S_2 \\ 0, & \text{otherwise.} \end{cases} \tag{3.11}$$

To consider the node labels and edge labels, we modified the graphlet kernel equation into

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in M_1^k} \sum_{S_2 \in M_2^k} k_{subgraph}(S_1, S_2) \tag{3.12}$$

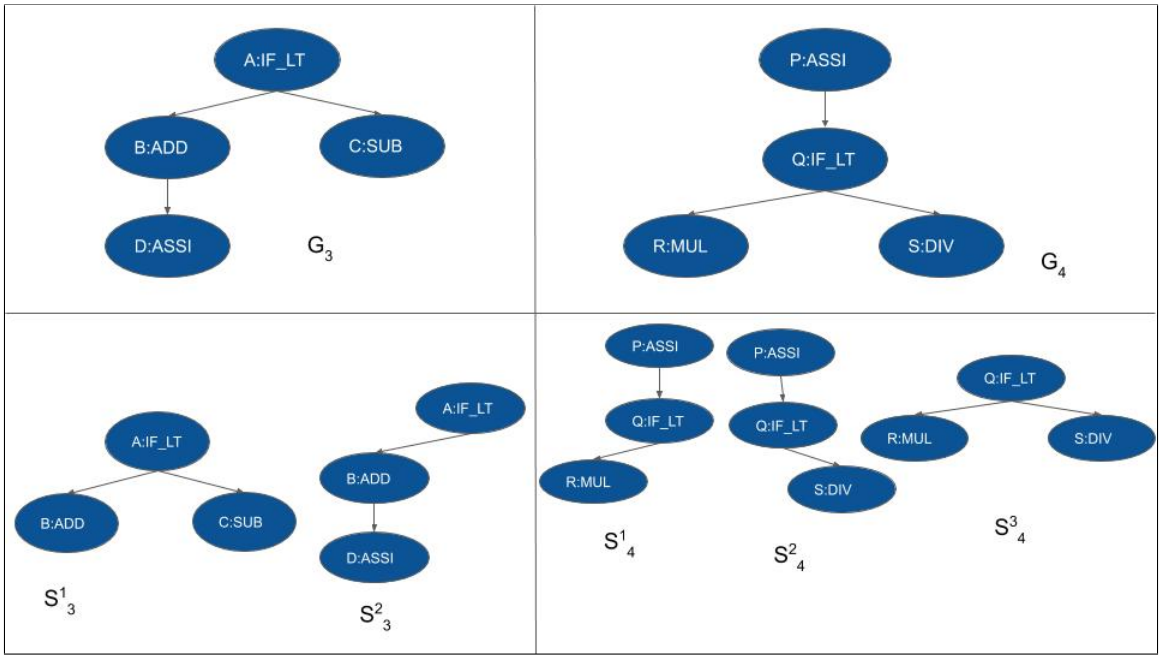

Figure 3.4: Graphlet kernel computation for graph G_3 and G_4

, where

$$k_{subgraph}(S_1, S_2) = \begin{cases} \prod_{v \in V_{s_1}} k_{node}(v, g(v)) * \prod_{(v_1, v_j) \in E_{s_1}} k_{edge}((v_i, v_j), (g(v_i), g(v_j))), & \text{if } S_1 \simeq S_2 \\ 0, & \text{otherwise.} \end{cases} \tag{3.13}$$

The graphlet kernel value for a pair of programs represented in the graph-based representation, that was described in this section, was computed using equation 3.12. Similar to the random walk kernel, k_{node} in equation 3.13 is computed as shown in equation 3.4 and k_{edge} is computed by equation 3.5.

For example, in Figure 3.4 G_3 and G_4 are graph representation of two functions. Suppose, the graphlet size of the graphs are restricted to size 3, considering G_3 and G_4, with graphlet size 3, the sub-graphs are- $S_3^1, S_3^2, S_4^1, S_4^2$, and S_4^3.

Then it computes the similarity score between two graphs using graphlets.

$$k_{\text{graphlet}}(G_3, G_4) = k_{\text{graphlet}}(S_3^1, S_4^1) + k_{\text{graphlet}}(S_3^1, S_4^2) + k_{\text{graphlet}}(S_3^1, S_4^3) + ... + k_{\text{graphlet}}(S_3^2, S_4^2) + k_{\text{graphlet}}(S_3^2, S_4^3)$$

Similarity between two graphlets are computed as follows-

$$k_{\text{graphlet}}(S_3^1, S_4^1) = 0.1$$

$$k_{\text{graphlet}}(S_3^1, S_4^2) = 0.1$$

$$k_{\text{graphlet}}(S_3^1, S_4^3) = k_{\text{node}}(A, Q) \times k_{\text{node}}(B, R) \times k_{node}(C, S) \times k_{\text{edge}}((A, B), (Q, R)) \times k_{\text{edge}}((A, C), (Q, S))$$

$$\vdots$$

$$k_{\text{graphlet}}(S_3^2, S_4^2) = k_{\text{node}}(A, P) \times k_{\text{node}}(B, Q) \times k_{node}(D, S) \times k_{\text{edge}}((A, B), (P, Q)) \times k_{\text{edge}}((B, D), (Q, S))$$

$$k_{\text{graphlet}}(S_3^2, S_4^3) = 0.1$$

Computation of the similarity score between two nodes and edges are shown below -

$$k_{\text{node}}(A, Q) = 1$$

$$k_{\text{node}}(B, R) = 0.6$$

$$k_{\text{node}}(C, S) = 0.6$$

$$\vdots$$

$$k_{\text{node}}(D, S) = 0.1$$

$$k_{\text{edge}}((A, B), (Q, R)) = 1$$

$$\vdots$$

$$k_{\text{edge}}((B, D), (Q, S)) = 1$$

This concept of *graphlet graph kernel* computation process is used in this thesis.

3.3.4 Predictive Model

As suggested before, the prediction of MRs is formulated as a machine learning problem. Machine learning algorithms are an effectively well-known procedure for determining linear relations. In this section, supervised learning is considered. There are many widely used algorithms for supervised learning; among them, kernel-based support vector machines are used in this thesis.

3.3.4.1 Support Vector Machines: In many real-world applications, SVM algorithms are used for classification or regression issues [36]. The basic concept of SVMs is to perform linear classification where it creates a hyperplane in the high dimensional space [36]. This hyperplane can distinguish examples of different classes in the training sets based on the information of their associated class labels [36]. Many accessible kernel functions such as the linear kernel, polynomial kernel, Gaussian kernel, and sigmoid kernel can perform the mapping of the original low dimensional data sets into a higher dimensional space [36]. As shown in Figure 3.5, SVMs find a hyperplane amongst the data points that separate the classes of data. A hyperplane

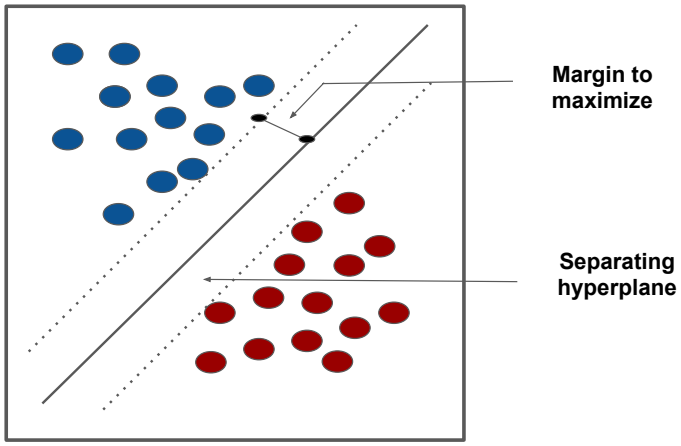

Figure 3.5: General classification hyperplane representation of SVM algorithm

maximizes the margin between itself and the support vectors; therefore, it decreases the classification error [36]. SVMs are inductive model, as it predict the testing sets based on mathematical models obtained from the training sets [36]. It creates a classification model or a classifier using labeled data. The model is used to predict labels for previously unseen data. It can also be used for binary classification [36].

In this thesis, the kernel function is the graph kernel functions, which are described in this section. The computed graph kernel values are supplied to SVMs with a binary label indicating whether a given function satisfies a given MR or not. The SVMs use the provided information to create a model that can predict if a new function would satisfy the considered MR or not. In this study, the SVMs implementation from the scikit-learn[2] toolkit was used. Figure 3.1 shows an overview of the creation of the predictive model.

[2]http://scikit-learn.org/stable/

3.4 Experimental setup

This section describes the code corpus and metamorphic relations used in this study. The details of the evaluation procedure are also discussed here.

3.4.1 Code corpus

A total of 93 functions that perform matrix calculations are used to measure the effectiveness of the proposed method for predicting MRs. They are collected from Apache Commons Math Library[3], la4j (Linear Algebra for Java)[4], and JAMA (Java Matrix package)[5], all of which are open-source projects. These functions execute a variety of calculations on matrices such as addition, multiplication, subtraction, and searching (e.g., getting column matrix, getting row matrix). Many of the functions have the same functionality but are implemented differently. For example, Array2DRowRealMatrix class and OpenMapRealMatrix class both have multiplication functions for matrices, but they are implemented in different ways. In such cases, both functions are used in the code corpus. All the functions used in this study can be found in Appendix A.

The proposed approach leverages the graph kernel method that calculates the similarity score between a pair of graphs; therefore, CFGs are generated for these functions using the Soot framework. Each node in the generated CFG is labeled to specify its corresponding operation.

3.4.2 Metamorphic Relations

For this study, ten MRs are identified manually that are generally applicable to matrix calculations. These MRs are used as class labels for the classification model.

[3]http://commons.apache.org/proper/commons-math/javadocs/api-3.6/
[4]http://la4j.org/apidocs/
[5]https://math.nist.gov/javanumerics/jama/doc/

The following are the list of MRs identified for this study-

1. MR1 - ScalarAddition: If a positive constant is added to the positive source input matrix to create a follow-up input matrix, the summation of elements of the follow-up output matrix must be greater than or equal to the summation of elements of the source output matrix. For example, A is an input matrix, and b is a positive constant. For MR1, the follow-up input is $A^{'}$, where $\forall i, j, a^{'}_{i,j} = a_{i,j} + b$; therefore, the expected relation among the follow-up and the source output is $\sum_i \sum_j o^{'}_{(i,j)} \geq \sum_i \sum_j o_{(i,j)}$.

2. MR2 - AdditionWithIdentityMatrix: If an identity matrix, similar in size to the positive source input matrix is added to generate a follow-up input, the summation of elements of the follow-up output matrix must be greater than or equal to the summation of elements of the source output matrix. For example, A is an input matrix, and I is an identity matrix. For MR2, the follow-up input is A', where $\forall i, j, a^{'}_{i,j} = i_{i,j} + a_{i,j}$; therefore, the expected output relation is $\sum_i \sum_j o^{'}_{(i,j)} \geq \sum_i \sum_j o_{(i,j)}$.

3. MR3 - ScalarMultiplication: If a positive constant is multiplied to the positive source input matrix to create a follow-up input matrix, the summation of elements of the follow-up output matrix must be greater than or equal to the summation of elements of the source output matrix. For example, A is an input matrix, and b is a positive constant. For MR3, the follow-up input is $A^{'}$, where $\forall i, j, a^{'}_{i,j} = b \times a_{i,j}$; therefore, expected output relation is $\sum_i \sum_j o^{'}_{(i,j)} \geq \sum_i \sum_j o_{(i,j)}$.

4. MR4 -MultiplicationWithIdentityMatrix: If an identity matrix, similar in size to the positive source input matrix is multiplied element by element to generate

a follow-up input, the summation of elements of the follow-up output matrix must be less than or equal to the summation of elements of the source output matrix. For example, A is an input matrix, and I is an identity matrix. For MR4, the follow-up input is A', where $\forall i, j, a'_{i,j} = i_{i,j} \times a_{i,j}$; therefore, the expected output relation is $\sum_i \sum_j o'_{(i,j)} \leq \sum_i \sum_j o_{(i,j)}$.

5. MR5 - Transpose: If a follow-up input is generated by transposing the positive source input matrix, the summation of elements of the follow-up output matrix must be equal to the summation of elements of the source output matrix. For example, A is an input matrix. For MR5, the follow-up input is A', where $\forall i, j, a'_{i,j} = a_{j,i}$; therefore, the expected output relation is $\sum_i \sum_j o'_{(i,j)} = \sum_i \sum_j o_{(i,j)}$.

6. MR6 - MatrixAddition: If the positive source input matrix is added to itself, to generate the follow-up input, the summation of elements of the follow-up output matrix must be greater than or equal to the summation of elements of the source output matrix. For example, A is an input matrix. For MR6, the follow-up input is $A' = A + A$; therefore, expected output relation is $\sum_i \sum_j o'_{(i,j)} \geq \sum_i \sum_j o_{(i,j)}$.

7. MR7 - MatrixMultiplication: If the positive source input matrix is multiplied to itself, to generate the follow-up input, the summation of elements of the follow-up output matrix must be greater than or equal to the summation of elements of the source output matrix. For example, A is an input matrix. For MR7, the follow-up input is $A' = A \times A$; therefore, the expected output relation is $\sum_i \sum_j o'_{(i,j)} \geq \sum_i \sum_j o_{(i,j)}$.

8. MR8 - PermuteColumn: If the columns of the positive source input matrix is permuted to create the follow-up input, the summation of elements of the

follow-up output matrix must be equal to the summation of elements of the source output matrix. . For example, A is a input matrix with $j = 1, 2, 3, .., n$ columns. For MR8, the follow-up test case is A' after permuting the column positions; therefore, the expected output relation is $\sum_i \sum_j o'_{(i,j)} = \sum_i \sum_j o_{(i,j)}$.

9. MR9 - PermuteRow: If the rows of the positive source input matrix is permuted to create the follow-up input, the summation of elements of the follow-up output matrix must be equal to the summation of elements of the source output matrix. . For example, A is a input matrix with $j = 1, 2, 3, .., n$ rows. For MR8, the follow-up test case is A' after permuting the row positions; therefore, the expected output relation is $\sum_i \sum_j o'_{(i,j)} = \sum_i \sum_j o_{(i,j)}$.

10. MR10 - PermuteElement: If the elements of the positive source input matrix is permuted to create the follow-up input, the summation of elements of the follow-up output matrix must be equal to the summation of elements of the source output matrix. . For example, A is a input matrix with $i = 1, 2, 3, .., n$ rows and $j = 1, 2, 3, .., n$ columns. For MR10, the follow-up test case is A' after permuting elements. Therefore, the expected output relation is $\sum_i \sum_j o'_{(i,j)} = \sum_i \sum_j o_{(i,j)}$.

We manually identified whether the programs in our code corpus satisfy these 10 MRs or not. Moreover, one of my other lab partners helped me verifying the identification of the MRs by rechecking them. It took a fair amount of time to manually come up with the MRs and identify them for the code corpus of this study. Table 3.1 shows the number of positive and negative instances for each MR; positive indicates that a function satisfies the given MR and negative indicates that the function does not satisfy the given MR.

Metamorphic Relation	Positive instances	Negative instances
MR1	77	16
MR2	25	68
MR3	77	16
MR4	39	54
MR5	36	57
MR6	12	81
MR7	20	73
MR8	55	38
MR9	55	38
MR10	55	38

Table 3.1: Number of positive and negative instances for each metamorphic relation

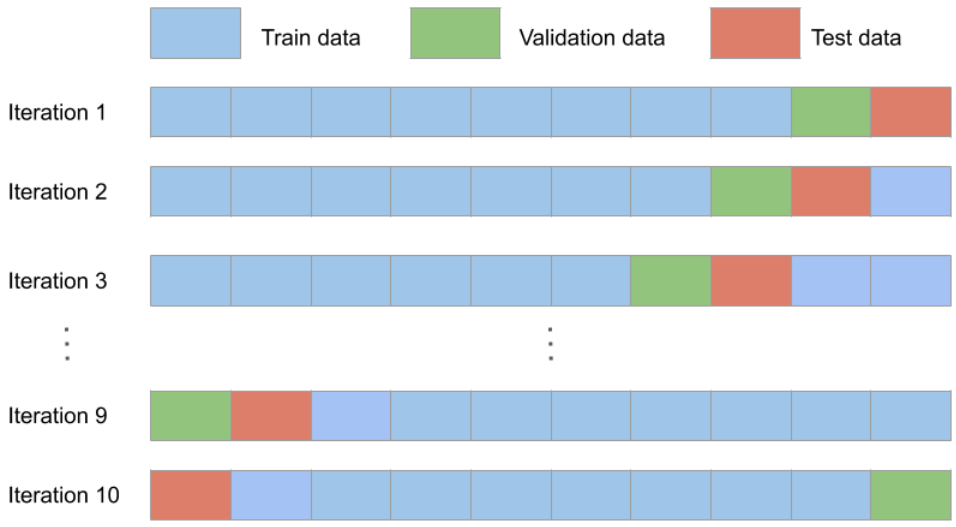

Figure 3.6: Representation of the *stratified train, validation and test* setup for SVMs model

3.4.3 Evaluation Procedure

First of all, I normalize each kernel such that each example has a unit norm by the expression using cosine normalization [37]. Following is the equation of the normalization used in this study,

$$k'_{graph}(G_1, G_2) = \frac{k_{graph}(G_1, G_2)}{\sqrt{k_{graph}(G_1, G_1) k_{graph}(G_2, G_2)}}$$

We use stratified *train, validation, and test setup* to evaluate the MR prediction effectiveness. Figure 3.6 illustrates the method. For each MR, we divided the data into 10 folds, where each fold contained approximately the same portion of positive and negative instances, as the original data set. The folds are divided into three subsets. The three sets were named as Train data, Test data, and Validation data. The precomputed kernel values of the functions in Train data are used to create the prediction model. The Validation data was used to select parameters for the predictive model. The parameters are- (1) regularization parameter C of the SVMs (2) path weighing factor λ in the random walk kernel where $0 \leq \lambda < 1$ (i.e., each walk of length n is weighted by λ^n, where n ranged from 1 to 10) (3) sub-graph size (2,3,4) in graphlet kernel. The parameter values selected using the validation set were then used to create the binary predictive model for predicting the MRs for the test data. We repeated the train, validation, and test method ten times so that each time each fold can be used for validation and testing while the rest of the folds are used for training the model to avoid any biases occurring in fold divisions. Therefore, every time eight folds are used for training, one fold is used for validation, and one fold is used for testing.

Metamorphic Relation	Best C	Best λ
MR1	0.1	0.5
MR2	10	0.9
MR3	1000	0.7, 0.6
MR4	100, 1000	0.7
MR5	10, 100, 1000	0.8, 0.9
MR6	1	0.8, 0.9
MR7	1000	0.5
MR8	1000	0.3
MR9	1000	0.9, 0.3
MR10	1000	0.7, 0.8

Table 3.2: Best C and best λ parameter values for selecting predictive model for each metamorphic relation on validation set

3.4.4 Area Under the Receiver Operating Characteristic Curve

The evaluation measure used in this study is *Area Under the receiver operating characteristic Curve (AUC)*. AUC measures the probability that a randomly chosen positive example (i.e., a program labeled with a certain MR) will be ranked higher by the predictive model than a randomly chosen negative example [38]. AUC takes values ranged in [0,1] where higher values indicate better performance, but a value of 0.5 is equivalent to a random classifier. AUC is used as the evaluation metric in this experiment as it does not depend on the discrimination threshold of the classifier and is considered a better measure for analyzing learning algorithms (compared to metrics such as accuracy) [38].

Metamorphic Relation	Best C	Best graphlet size
MR1	100	3
MR2	1	3
MR3	0.1, 1, 10, 100	2
MR4	100, 1000	3
MR5	10	2
MR6	100	2
MR7	100	2
MR8	0.1	3
MR9	1000	2
MR10	100	2

Table 3.3: Best C and best graphlet size parameter values for selecting predictive model for each metamorphic relation on validation set

3.5 Results

The performance of the random walk kernel and graphlet kernel are analyzed using SVMs for building the predictive model.

3.5.1 Predictive Model Selection

The validation set is used to select the set of parameter values for identifying the best predictive model. Table 3.2 and 3.3 lists the λ, graphlet size, and C values for every MR on the validation set, that corresponded to the highest AUC value. For the MRs in Table 3.2, the value selected for the parameter C does not seem to have a big effect on the prediction accuracy as most of them has a value of 1000, except MR1, MR2 and MR3. But the best value for λ is different for different MRs. 7 out of the 10 MRs (i,e., MR2, MR3, MR4, MR5, MR6, MR9 and MR10) has the best λ value greater than 0.5. When the λ value is higher, the random walk kernel gives a higher weight to longer paths according to it's definition [4]. However, 4 MRs (i.e., MR1,MR7, MR8 and MR9) has the best λ value between 0.3 to 0.5. Therefore this indicates that for these 4 MRs, longer paths in the CFGs are more important for predicting MRs than the other paths.

For the graphlet kernel, the value selected for the parameter C is different for all the MRs. The best C value of 100 is selected for MR1, MR3, MR4, MR6, MR7, and MR10. However, MR2, MR5, MR8, and MR9 have different best C values. This indicates that the parameter C has an effect on the prediction accuracy. Also, the best size of a graphlet is different for different MRs. 6 out of the 10 MRs (i,e., MR3, MR5, MR6, MR7, MR9, and MR10) has a value of 2 for the best graphlet size. Moreover, 4 MRs (i.e., MR1, MR2, MR4, and MR8) has a value of 3 for best graphlet size.

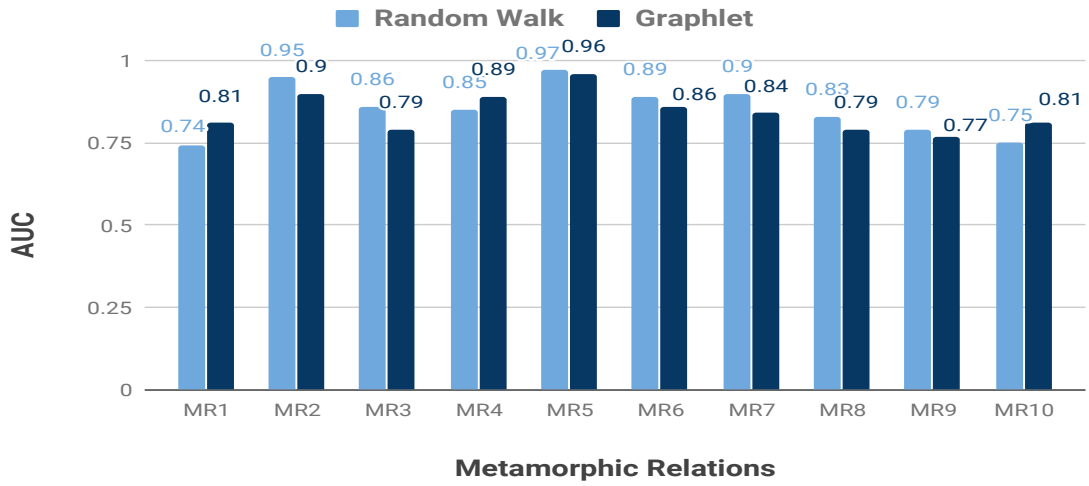

Figure 3.7: Prediction *AUC* score for *Random walk graph kernel* and *Graphlet kernel* on test set using the predictive model with best parameters

3.5.2 Comparison Between the Graph Kernels

Random walk kernel and graphlet kernel, two types of graph kernels, are used to train the predictive model. Figure 3.7 shows the AUC scores for the test data set with both random walk kernel and graphlet kernel. Among the 10 MRs, 7 MRs (i,e., MR2, MR3, MR5, MR6, MR7, MR8, and MR9) performs better when using the random walk kernel. For MR1, MR4, and MR10, the graphlet kernel performs better. The highest AUC score of 0.97 with the random walk kernel is observed when predicting the MR5. Table 3.1 shows the ratio of positive and negative instances for each MR. However, the result shows that the ratio has an effect on the prediction accuracy of the MRs. Here, MR2, MR5, and MR7 consist of low positive and high negative instances when the prediction AUC scores are higher than 0.9 using random walk kernel. The other MRs also reported AUC values higher than 0.73, indicating that the approach has created effective predictive models for all the MRs. An MR is a property that specifies the relation between the outputs, which are generated by

multiple executions of the function, which in-turn might correlate with the MRs. In this study, the result shows that the random walk kernel that uses execution traces for the comparison of two functions performs better than the graphlet kernel.

CHAPTER FOUR

USING SEMI-SUPERVISED LEARNING TO PREDICT METAMORPHIC RELATION

4.1 Approach

In this section, the graph kernel-based machine learning method is extended, where a semi-supervised classification algorithm is used to build the predictive model. The model is build using the approach showed in Figure 3.1 of Chapter 3. Also, a comparison is performed between the learning algorithms and graph kernels to evaluate the proposed method.

4.1.1 Predictive Model

To evaluate the performance of the suggested method by incorporating unlabelled data to build a predictive model, semi-supervised learning is considered. There are multiple algorithms for semi-supervised learning, among them a kernel-based semi-supervised support vector machines (S3VMs) are used in this thesis.

4.1.1.1 Semi-Supervised Support Vector Machines (S3VMs)

For the semi-supervised model, the semi-supervised support vector machines (S3VMs) are used [39]. In 1999, the semi-supervised support vector machine was first introduced by Bennett and Demiriz [39]. S3VM and SVM are almost similar, except in two essential features. Firstly, SVMs only accepts labeled data, whereas S3VMs accepts unlabeled data with some labeled data. Secondly, S3VM, aka Transductive Support Vector Machines, is a transductive model, as it predicts testing sets based on specific data patterns seen in training sets [39]. The S3VM algorithm generates a prediction function to predict labels for previously unseen data points [39]. Additional constraints are included for

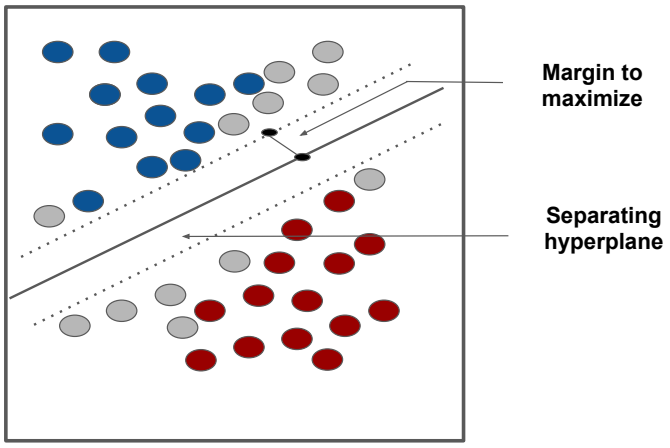

Figure 4.1: General classification hyperplane representation of Semi-Supervised Support Vector Machine (S3VM) algorithm

optimization, which calculates the misclassification errors for each data point and class, and finally, the algorithm selects the class with the minimum misclassification error [39]. Figure 3.5 shows the hyperplane derived without the unlabeled data using SVM. Figure 4.1 shows that adding unlabeled data can increase the classification accuracy using S3VM. The implementation called QN-S3VM, designed by Gieseke et al. [40], is used for the experiment. This algorithm is selected over other S3VMs as it can handle various types of kernels. In a previous research work of our group, the code [40] was revised to read in precomputed kernels such as random walk and graphlet kernel.

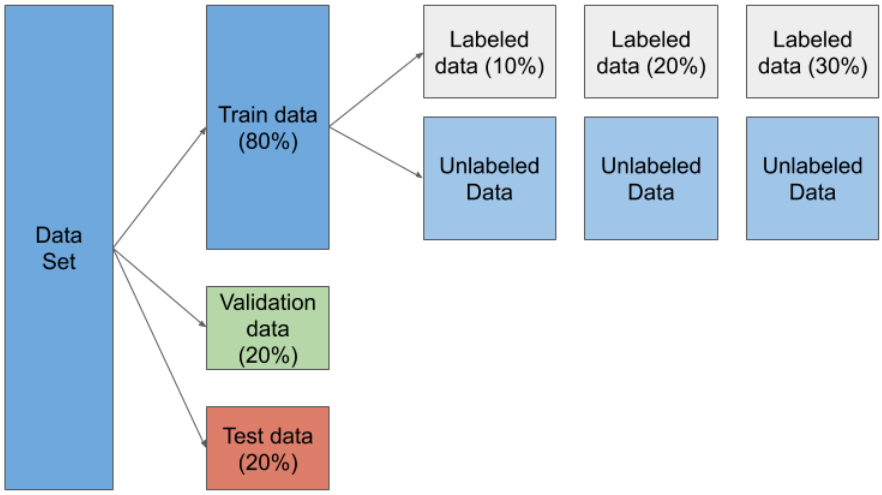

Figure 4.2: Representation of the *stratified train, validation and test* setup using S3VM

4.2 Experimental Setup

The experiment uses the same code corpus, and MRs described in Chapter 3. Though the same evaluation metric, AUC is used in this experiment, the evaluation procedure is different. This experiment presents a comparison between the supervised and semi-supervised classification models, as well as between the graph kernels. This section describes the details of the evaluation procedure for the predictive model.

4.2.1 Evaluation procedure

We use the stratified *train, validation, and test setup* to evaluate the MR prediction effectiveness among the classification models. The number of positive and negative instances for each MR is the same as in Chapter 3. Figure 4.2 illustrates the method. For each MR, we divided the data into three subsets, where each set

contained approximately the same portion of positive and negative instances, as the original data set. The three sets are- (1) Train data (i.e., 80% of the data), (2) Test data (i.e., 10% of the data), and (3) Validation data (i.e., 10% of the data). The Train data is further divided into two parts called the labeled data and unlabelled data. The supervised and semi-supervised predictive models are trained using 10%, 20% and 30% of the labeled data each time. For the semi-supervised model, the rest of the training data are used as the unlabelled data. The Validation data is used to select parameters for the predictive models. The parameters are- (1) regularization parameter C (in case of SVM only) (2) path weighing factor λ in the random walk kernel where $0 \leq \lambda < 1$ (3) sub-graph size (2,3,4) in graphlet kernel. The parameter values selected using the validation set are then used to create the predictive models for predicting the MRs for the test data. This process is repeated ten times so that the functions in the labeled data set are selected randomly each time to avoid any biases that occur in train data divisions of labeled and unlabelled data.

4.3 Results

The graph kernel-based machine learning approach is analysed in two ways. Firstly, a comparison is done based on the two types of machine learning techniques, SVM and S3VM. Secondly, the performance of the random walk kernel and graphlet kernel are analyzed using only the S3VM for the predictive model.

Metamorphic Relations	labeled data(%)	Best λ-SVM	Best C-SVM	Best λ-S3VM
MR1	30	0.9	10	0.3
	20	0.2,0.3,0.5	1,10,1000	0.4,0.5
	10	0.5,0.6,0.7	0.1,1,10,100,1000	0.9

MR2	30	0.2	0.1	0.6
	20	0.1,0.6,0.7,0.8	0.1,1,10,100,1000	0.8
	10	0.1	0.1	0.8
MR3	30	0.1,0.2,0.3,0.4	1000	0.3
	20	0.1,0.4	0.1,1	0.9
	10	0.1	1	0.9
MR4	30	0.1	1000	0.5
	20	0.2	1000	0.8
	10	0.1	1000	0.2, 0.9
MR5	30	0.2	0.1	0.9
	20	0.3	1000	0.6
	10	0.1	0.1	0.2
MR6	30	0.1	0.1	0.1
	20	0.1	0.1	0.1
	10	0.1,0.2,0.3,0.4,0.5	0.1,1,10,100,1000	0.6,0.7,0.9
MR7	30	0.1	1000	0.5
	20	0.1	1000	0.9
	10	0.9	10, 100, 1000	0.4
MR8	30	0.7	100, 1000	0.8
	20	0.8	100, 1000	0.8,0.9
	10	0.8,0.9	0.1, 100, 1000	0.7
MR9	30	0.5	100	0.5
	20	0.9	10, 100, 1000	0.9
	10	0.4,0.5,0.6,0.7	0.1,1,10,100,1000	0.7
MR10	30	0.8	100, 1000	0.1

	20	0.1,0.5	100, 1000	0.8
	10	0.3	1000	0.6

Table 4.1: Best C and best λ parameter values for selecting predictive model for each metamorphic relation on validation set with different ranges of labeled data (i.e., 10%, 20%, and 30%). For SVM based model uses both C and λ parameters. S3VM based model uses only the λ parameter.

4.3.1 Predictive Model Selection

The validation set is used to select the set of parameter values for identifying the best predictive model. Table 4.1 lists the best λ for SVM and S3VM classification models and C values for the SVM classification model, for which the highest AUC values for each MR on the validation set was recorded. Here, the value selected for C has an effect on the prediction accuracy for all of the MRs. The most common value selected for C, among all the MRs is 1000. The best value for λ is different for different MRs and classification models. The most selected value recorded for the best λ is 0.1 for MR2, MR3, MR4, MR5, MR6, MR7, and MR10 when using the SVM for the predictive model. On the other hand, when using S3VM as the predictive model, the most selected best λ value is 0.9 for MR1, MR3, MR4, MR5, MR6, MR7, MR8, and MR9.

Table 4.2 lists the best graphlet size for SVM and S3VM classification models and C values for the SVM classification model, for which the highest AUC values for each MR on the validation set were recorded. In the graphlet kernel, for all the MRs, the most selected best C values are 1000, which indicates that the parameter C has an effect on the prediction accuracy. Also, the common best size of a graphlet is 4 for MR1, MR2, MR3, MR4, MR8, MR9, and MR10, when using SVM. For S3VM, 2 is the most selected best graphlet size for MR1, MR2, MR3, MR6, MR7, MR8, and MR9.

Metamorphic Relations	labeled data(%)	Best graphlet size-SVM	Best C-SVM	Best graphlet size-S3VM
MR1	30	4	100, 1000	4
	20	2	1000	2
	10	4	0.1,1,10,100,1000	2
MR2	30	4	0.1,1,10,100	2,3
	20	3,4	10, 100, 1000	4
	10	3	0.1,1,10,100,1000	2
MR3	30	3	100	3,4
	20	4	0.1,1,10,100,1000	2,3,4
	10	2,3	0.1,1,10,100,1000	2
MR4	30	4	0.1,1,10,100,1000	3
	20	4	0.1,1,10,100,1000	3
	10	4	0.1,1,10,100,1000	3
MR5	30	3	100, 1000	4
	20	3	10 , 100 , 1000	3
	10	2, 3	0.1,1,10,100,1000	4
MR6	30	2	10, 100, 1000	3
	20	3	10, 100, 1000	3
	10	2	0.1,1,10,100,1000	2
MR7	30	3	1000	2
	20	3	10, 100, 1000	2
	10	3	10, 100, 1000	2
MR8	30	4	100, 1000	2

	20	4	1	2
	10	2	10	4
	30	4	100, 1000	4
MR9	20	4	1	4
	10	3	10	2,3,4
	30	2	100, 1000	4
MR10	20	4	1	4
	10	2	10	3

Table 4.2: Best C and best graphlet size parameter values for selecting predictive model for each metamorphic relation on validation set with different ranges of labeled data (i.e., 10%, 20%, and 30%). For SVM based model uses both C and graphlet size parameters. S3VM based model uses only the graphlet size parameter.

4.3.2 Comparisons between the Machine Learning Techniques

SVM and S3VM, two types of machine learning techniques are used to create a predictive model. Figure 4.3 shows the AUC scores of both SVM and S3VM predictive models with 10%, 20%, and 30% labeled data, respectively, for the test data set using the random walk kernel. For 10% and 20% labeled data, 5/10 MRs (i,e., MR1, MR2, MR3, MR5, and MR8) perform better when using S3VM. MR7, for 10% labeled data, and MR4 and MR6 for 20% labeled data, perform the same for both the models. For 30% labeled data, 6/10 MRs (i,e., MR1, MR2, MR3, MR4, MR5, and MR6) perform better when using S3VM. The highest AUC score with S3VM is observed when predicting MR1 (0.82), MR2 (0.88), and MR3 (0.88) for 10% , 20% , and 30% labeled data. From the results, it can be observed that MR1, MR2, MR3, and MR5 are predicted with high AUC across the different ranges of labeled data when using S3VM. However, the ratio of positive and negative instances does not have an effect on prediction accuracy. Here, MR1 has high positive instances, but MR2 has

low positive instances. Again, MR1 has low negative instances, but MR2 has high negative instances.

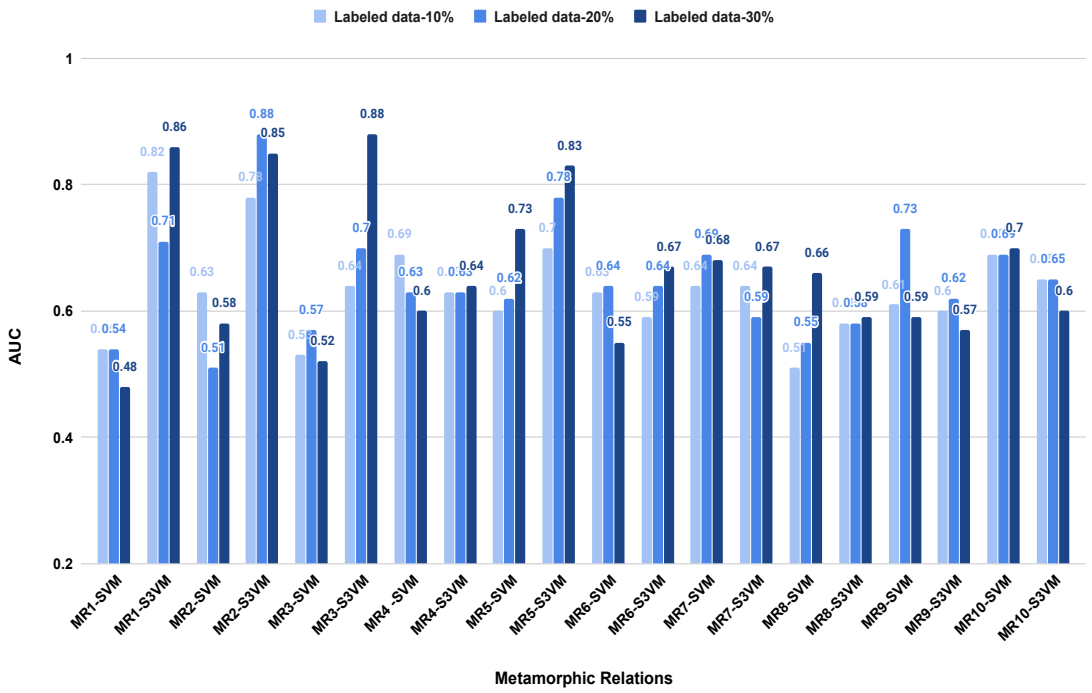

Figure 4.3: Prediction *AUC* score for each metamorphic relation on the *random walk kernel*-based model with best parameters using 10%, 20%, and 30% labeled data

Figure 4.4 shows the AUC scores of both SVM and S3VM predictive model with 10%, 20% and 30% labeled data, respectively, for the test data set using the graphlet kernel. The result shows that for 10% labeled data, MR1, MR2, MR3, MR4, MR5, MR6 and MR7, for 20% labeled data, MR1, MR2, MR3, MR4, MR5, MR7 and MR9, and finally for 30% labeled data, MR1, MR2, MR3, MR4, and MR5 perform better when using S3VM. Using 30% labeled data, MR6 and MR7 perform similarly for both models. For graphlet kernel, the highest AUC score with S3VM is observed when predicting the MR7 (0.78), MR3 (0.76), and MR5 (0.93) for 10%, 20%, and 30% labeled data. From the results, it can be observed that MR1, MR2, MR3, MR4,

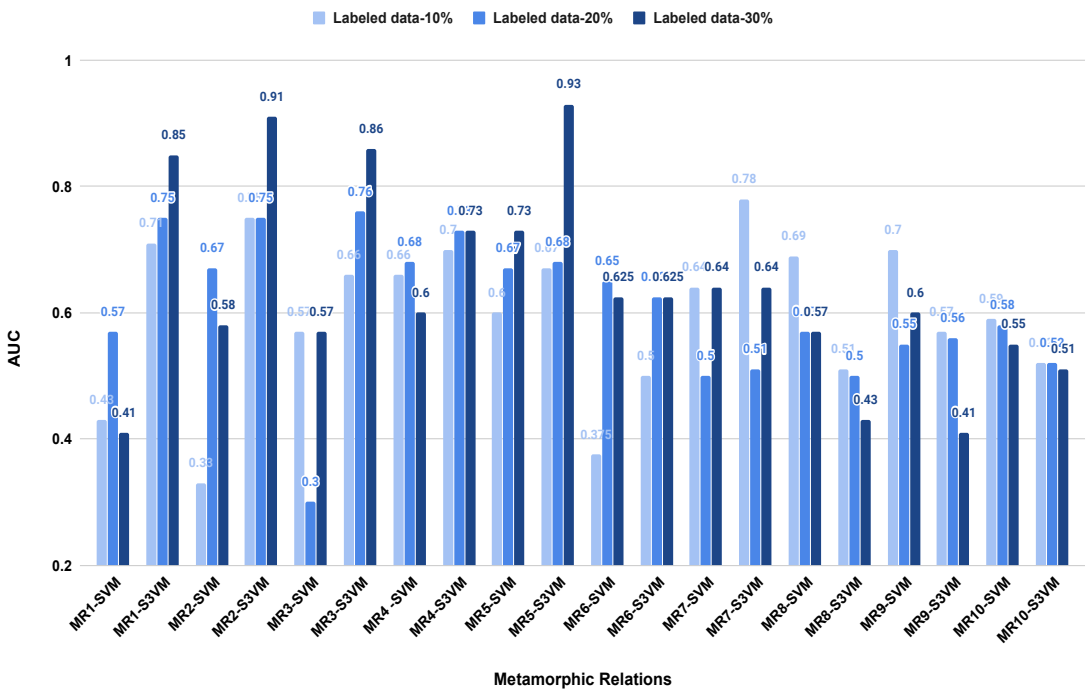

Figure 4.4: Prediction *AUC* score for each metamorphic relation on the *graphlet kernel*-based model with best parameters using 10%, 20%, and 30% labeled data

and MR5 are predicted with high AUC across the different ranges of labeled data when using S3VM. Here, the ratio of the positive and negative instances does not have an effect on the accuracy of the MRs.

In addition, a one tailed paired t-test is performed to determine the statistical significance of the improvement in the AUC scores, and the results comparing SVM and S3VM are shown in Tables 4.3 and 4.4. For the random walk kernel, MR1, MR2, MR5 and MR10 have p-values of less than 0.05, representing a statistically significant change. For the graphlet kernel, MR1, MR8 and MR10 have p-values less than 0.05. The MRs who have p-values greater than 0.05, represent no significance change in AUC scores. This indicates that adding more unlabelled data can improve the accuracy scores of a model therefore, this approach can create effective semi-

Metamorphic Relations	p-value
MR1	0.022
MR2	0.024
MR3	0.065
MR4	0.408
MR5	0.013
MR6	0.317
MR7	0.186
MR8	0.407
MR9	0.153
MR10	0.034

Table 4.3: T-test comparing SVM and S3VM based models using *random walk kernel* supervised predictive models for all the MRs.

Metamorphic Relation	p-value
MR1	0.029
MR2	0.056
MR3	0.060
MR4	0.061
MR5	0.118
MR6	0.273
MR7	0.191
MR8	0.028
MR9	0.111
MR10	0.011

Table 4.4: T-test comparing SVM and S3VM based models using *graphlet kernel*

4.3.3 Comparisons between the Graph Kernels

Random walk kernel and graphlet kernel, two types of graph kernels are used to train the predictive model. Figure 4.5 shows the AUC scores of the S3VM predictive model with 10%, 20%, and 30% labeled data, respectively, for the test data set using both the random walk kernel and graphlet kernel. Here, 7/10 MRs perform better, but a different group every time, when using 10% (i,e., MR1, MR2, MR5, MR6, MR8, MR9, and MR10), 20% (i,e., MR2, MR5, MR6, MR7, MR8, MR9, and MR10) and 30% (i,e., MR1, MR3, MR6, MR7, MR8, MR9, and MR10) labeled data using the random walk kernel. The highest AUC score with the random walk kernel is observed when predicting the MR1 (0.82), MR2 (0.88), and MR3 (0.88) for 10%, 20%, and 30% labeled data. Our results show that the random walk kernel performs better than the graphlet kernel for most of the MRs. They reported AUC values higher than 0.5, indicating that the approach created effective predictive models using the random walk kernel for all the MRs. In this experiment, the result shows again, that the random walk kernel that uses execution traces for the comparison of two functions, perform better than the graphlet kernel.

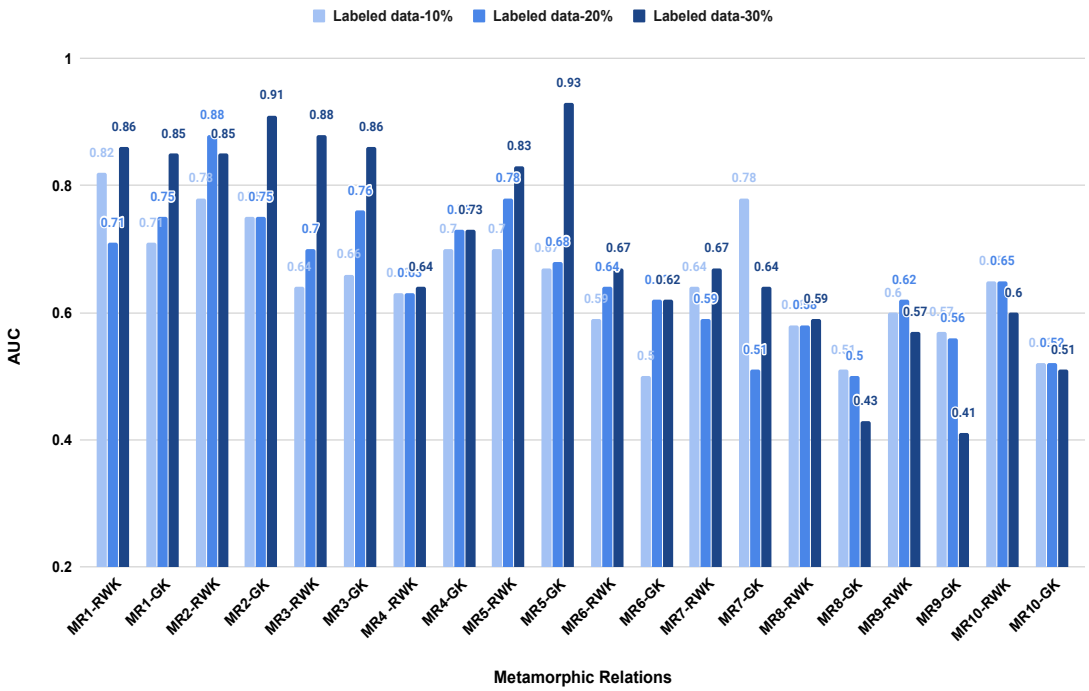

Figure 4.5: Prediction *AUC* score for each metamorphic relation on the *random walk kernel* (RWK) and *graphlet kernel* (GK) based S3VM models with best parameters using 10%, 20%, and 30% labeled data

CHAPTER FIVE

USING CALL GRAPH REPRESENTATION FOR METAMORPHIC RELATION PREDICTION

5.1 Approach

In this section, the graph kernel-based machine learning method described in Chapter 3 is extended, by using a program representation called *Call Graphs* , along with the CFG to build the predictive model. The model is build using the same approach shown in Figure 3.1 of Chapter 3, along with some extra computation. We only random walk kernel to compute the similarity scores using this graph representation since it performed better in our previous experiments compared to the graphlet kernel. The computation of the random walk kernel is different when CGs are incorporated with the CFGs. In this section, an introduction of the CGs is given. Also, it explains the modified random walk kernel approach used to build the predictive model. Both SVMs and S3VMs machine learning algorithms described in the previous chapters are also used to build the machine learning models.

5.1.1 Function Representation - Call Graphs (CGs)

In a Call Graph, a node represents each method, and an edge represents each function call [27]. A call graph can also be considered as a control flow graph that represents calling relationships between other programs in a system. A CG is a directed graph $CG_f = (V, E)$ of a function *f*. Let, x be a function or method called by *f*. Each such function call is represented by a node $v_x \in V$. As x is called by *f*, it can then be said that there is an edge e where $e = (v_f, v_x) \in E$. Function call flows of f is represented by all the edges in the call graph of the function [27]. We use the

Soot[1] framework to create the CGs. Figure 5.1 represents a function for calculating the determinant of a matrix and its CG representation. It represents an example of a CG used in this experiment, where each node is a method, each edge is the call between the nodes in the graph, and the highlighted lines of the code are the method or function call.

5.1.2 Updated Random Walk Kernel for CGs

After creating the CG of the programs, the next step is to use a graph kernel to compute the similarity between the graphs. For computing the random walk kernel using the CGs of the subject functions, identification of the method calls of all the subject functions are needed, where each method call is also a function. First, CFGs are generated for the method call functions. Random walk kernel method described in Chapter 3 is used to compute the similarity matrix for the method call functions. After that, CGs are generated for the subject function of the experiment. It computes the similarity score between two graphs by summing up the similarity scores of all the pairs of walks, which is computed by multiplying the similarity scores of their corresponding step pairs [4]. The similarity score of a pair of steps is originally calculated by multiplying the similarity scores of node and edge pairs that make up the step [4].

Here, the node labels represent the name of the method call function, and the edge labels represent the flow of the method calls. Each node in the node pair represents the pair of the method call function. The similarity score of a node pair is assigned using the similarity matrix computed by the random walk kernel method using the CFGs representation of the method call functions. It looks for the similarity scores of the method call functions of the subject functions in the similarity matrix

[1]https://www.sable.mcgill.ca/soot/

```
public double determinant() {
    if (rows != columns) {
        throw new IllegalStateException("Can not
compute determinant of non-square matrix.");
    }
    if (rows == 0) {
        return 0.0;
    } else if (rows == 1) {
        return get(0, 0);
    } else if (rows == 2) {
        return get(0, 0) * get(1, 1) -
               get(0, 1) * get(1, 0);
    } else if (rows == 3) {
        return get(0, 0) * get(1, 1) * get(2, 2) +
               get(0, 1) * get(1, 2) * get(2, 0) +
               get(0, 2) * get(1, 0) * get(2, 1) -
               get(0, 2) * get(1, 1) * get(2, 0) -
               get(0, 1) * get(1, 0) * get(2, 2) -
               get(0, 0) * get(1, 2) * get(2, 1);
    }
MatrixDecompositor decompositor =
withDecompositor(LinearAlgebra.LU);
Matrix[] lup = decompositor.decompose();
Matrix u = lup[1];
Matrix p = lup[2];
double result = u.diagonalProduct();
int[] permutations = new int[p.rows()];
for (int i = 0; i < p.rows(); i++) {
   for (int j = 0; j < p.columns(); j++) {
      if (p.get(i, j) > 0.0) {
         permutations[i] = j;
         break;
      }
   }
}
int sign = 1;
for (int i = 0; i < permutations.length; i++) {
   for (int j = i + 1; j < permutations.length; j++) {
      if (permutations[j] < permutations[i]) {
         sign *= -1;
      }
   }
}
return sign * result; }
```

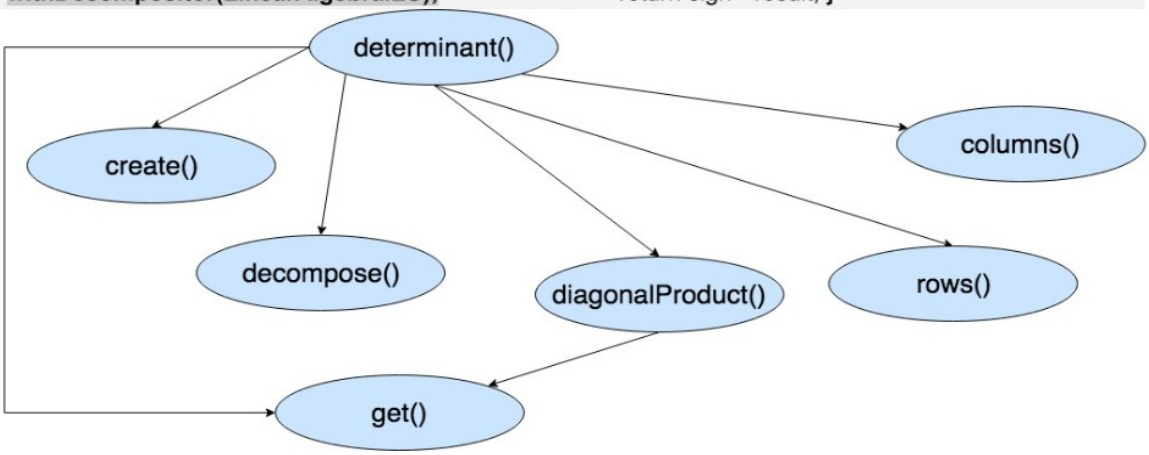

Figure 5.1: A function that computes the determinant of a matrix and its call graph (CG) representation

and assigns its value.

Edge labels also decide the value assigned for the similarity score of a pair of edges. Here, we only used one type of edge, which shows the flow of control between the method calls of the subject function. So the similarity score for a pair of edges is always one in regard to this study. The updated random walk kernel used in this study follows a similar definition of the random walk kernel, which is described in the previous work of Kanewala et al. [4]. Instead of assigning a specific value for the similarity of nodes, it assigns the similarity score of the method calls functions obtained using the random walk kernel.

5.1.2.0.1 Definition of the Updated Random Walk Kernel Suppose, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be call graph representations of two programs. Following Equation 3.1, we denote the kernel value of these two graphs as $k_{urw}(G_1, G_2)$, and as it is computed using the walks, the walk kernel is denoted as $k_{walk}(walk_1, walk_2)$ using the Equation 3.2 .

$$k_{urw}(G_1, G_2) = \sum_{walk_1 \in G_1} \sum_{walk_2 \in G_2} k_{walk}(walk_1, walk_2) \tag{5.1}$$

The walks consist of step kernels k_{step} defined in Equation 3.3, where the kernel for each step is defined using the kernel values of the two node pairs and the edge pair, as shown in Chapter 3. The node kernels are denoted as k_{node}, which is used to get the similarity score between two nodes. In the node kernel, we assign the similarity score by random walk kernel computed using control flow graphs of the programs in the nodes using Equation 5.3 as follows:

$$k_{node}(v_i, v_j) = k_{rw}(v_i, v_j), \tag{5.2}$$

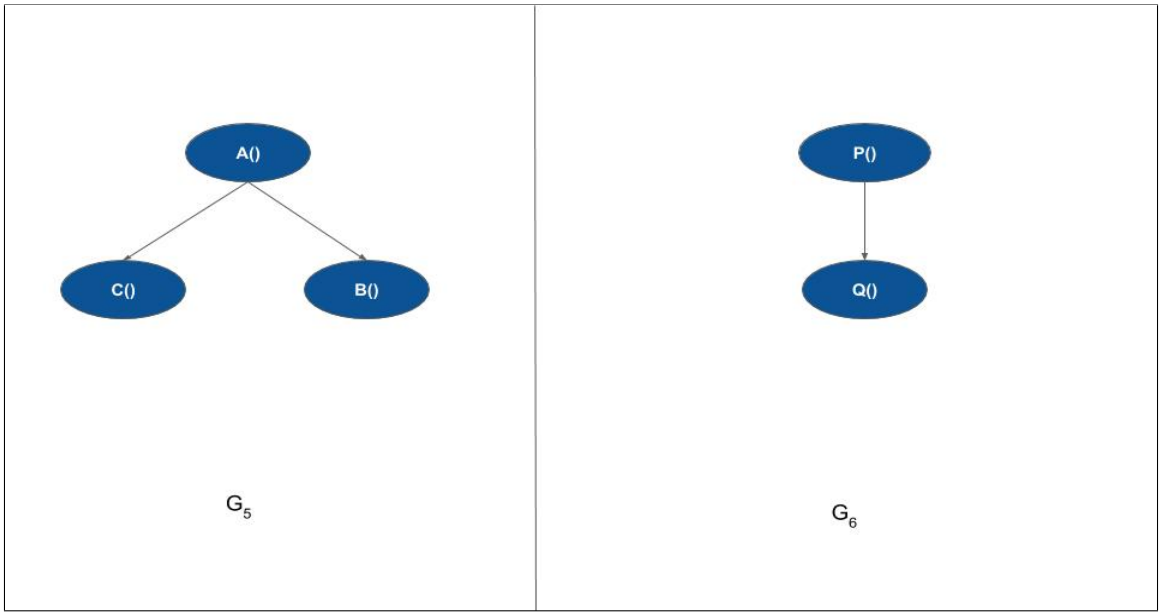

Figure 5.2: Updated random walk kernel computation for graph G_5 and G_6

where $v_i \in V_1$ and $v_j \in V_2$ represents the method call functions. However, the edge kernel k_{edge} is computed using the Equation 3.5. The direct product graph is then computed using Equations 3.6 and Equation 3.7. Based on the product graph, the updated random walk kernel, $k_{urm}(G_1, G_2)$ is defined following Equation 5.3.

$$k_{urw}(G_1, G_2) = \sum_{i,j=1}^{V_X} \left[\sum_{n=0}^{\infty} \lambda^n A_X^n \right]_{ij} \tag{5.3}$$

Here, A_X is denoted as the adjacency matrix of the direct product graph, where $1 > \lambda \geq 0$ is a weighting factor, and n is the path len.

For example, in Figure 5.2, G_5 and G_6 are graph representation of two functions, supposing the length of the walks of the graphs are restricted to one walks. Considering G_5, with walks of length one is

$$Length\ 1 : A \rightarrow B, A \rightarrow C$$

For G_6, with walk of length one is

$$Length\,1 : P \rightarrow Q$$

Then it computes the similarity score between the walks of the graphs.

$$k_{\text{walk}}(A \rightarrow B, P \rightarrow Q) = k_{\text{step}}((A, B), (P, Q))$$

$$k_{\text{walk}}(A \rightarrow C, P \rightarrow Q) = k_{\text{step}}((A, C), (P, Q))$$

Computation of the similarity between two steps are done as follows-

$$k_{\text{step}}((A, B), (P, Q)) = k_{\text{node}}(A, P) \times k_{\text{node}}(B, Q) \times k_{\text{edge}}((A, B), (P, Q))$$

$$k_{\text{step}}((A, C), (P, Q)) = k_{\text{node}}(A, P) \times k_{\text{node}}(C, Q) \times k_{\text{edge}}((A, C), (P, Q))$$

Similarity score between two nodes and edges are calculated as follows-

$k_{\text{node}}(A, P) =$ Random walk kernel value computed using CFGs of A and P

$k_{\text{node}}(B, Q) =$ Random walk kernel value computed using CFGs of B and Q

$k_{\text{node}}(C, Q) =$ Random walk kernel value computed using CFGs of C and Q

$$k_{\text{edge}}((A, B), (P, Q)) = 1$$

$$k_{\text{edge}}((A, C), (P, Q)) = 1$$

This concept of *random walk graph kernel* computation process is used in this experiment.

5.2 Experimental Setup

The ten MRs that are identified in Chapter 3 are used for this experiment. This section describes the code corpus and the details of the evaluation procedure.

5.2.1 Code Corpus

The experiment uses the same code corpus described in Chapter 3. However, few functions are eliminated as they do not have any method or function calls within them. So, 81 subject functions are used in this experiment. These subject functions have a total of 96 method call functions within them. All of these functions are collected from the same open-source projects mentioned in Chapter 3. All the functions used in this study can be found in Appendix A.

The proposed approach leverages the graph kernel method that calculates the similarity score between a pair of graphs; therefore, CFGs and CGs are generated for the method call functions and the subject functions, respectively.

5.2.2 Evaluation Procedure

The same evaluation metric, AUC, and the same evaluation procedure described in Chapter 3 and 4 are used in this experiment. This experiment presents the performance of supervised and semi-supervised classification models using only the random walk kernel, which is computed using both CFG and CG information of the programs.

The evaluation of the proposed approach is conducted for two scenarios. First, SVMs are used to build a predictive model. It uses the same *train, validation, and test* setup introduced in Chapter 3, where train data are used to create the prediction model, validation data is used to select parameters for the predictive model, and the test data are supplied into the best model to predict MRs. The parameters of selecting

the predictive model are the regularization parameter C and the path weighing factor λ in the random walk kernel, where $0 \leq \lambda < 1$ for CGs and CFGs. For a particular λ value when using call graphs for the subject functions, it looks into the similarity scores with all the λ values of the method call functions.

Secondly, S3VM is used to build the predictive model. It uses the same *train, validation, and test* setup described in Chapter 4. However, scenarios with 40% and 50% of labeled data are also included for evaluation purposes.

Metamorphic Relations	Best C	Control Flow Graph- Best λ	Call Graph- Best λ
MR1	1000	0.4	0.8
MR2	100	0.4	0.2
MR3	1000	0.6	0.3
MR4	10	0.7	0.9
MR5	1000	0.5	0.6
MR6	1	0.1	0.9
MR7	0.1	0.4	0.9
MR8	0.1	0.1	0.3
MR9	0.1	0.3	0.7
MR10	0.1	0.8	0.7

Table 5.1: Best C, best λ for CFGs, and best λ for CGs parameter values for each metamorphic relation for selecting predictive model on validation set

5.3 Results

The proposed approach is analyzed in two ways. Firstly, performance evaluation using SVMs, and secondly, using S3Vm.

5.3.1 Performance of SVMs

The validation set is used to select the set of parameter values for identifying the best predictive model. Table 5.1 lists the best λ for CGs, λ for CFGs, and C values for the SVM classification model, for which the highest AUC values for each MR on the validation set are recorded. Here, the value selected for C has an effect on the prediction accuracy for all of the MRs. The most common value chosen for C is 0.1 for MR7, MR8, MR9, and MR10. The best value of λ for CFGs and CGs are different for different MRs. The most selected value recorded for the best λ for CFGs is 0.4 for MR1, MR2, and MR7. However, for CGs, the most common best λ value is 0.9 for MR4, MR6, and MR7.

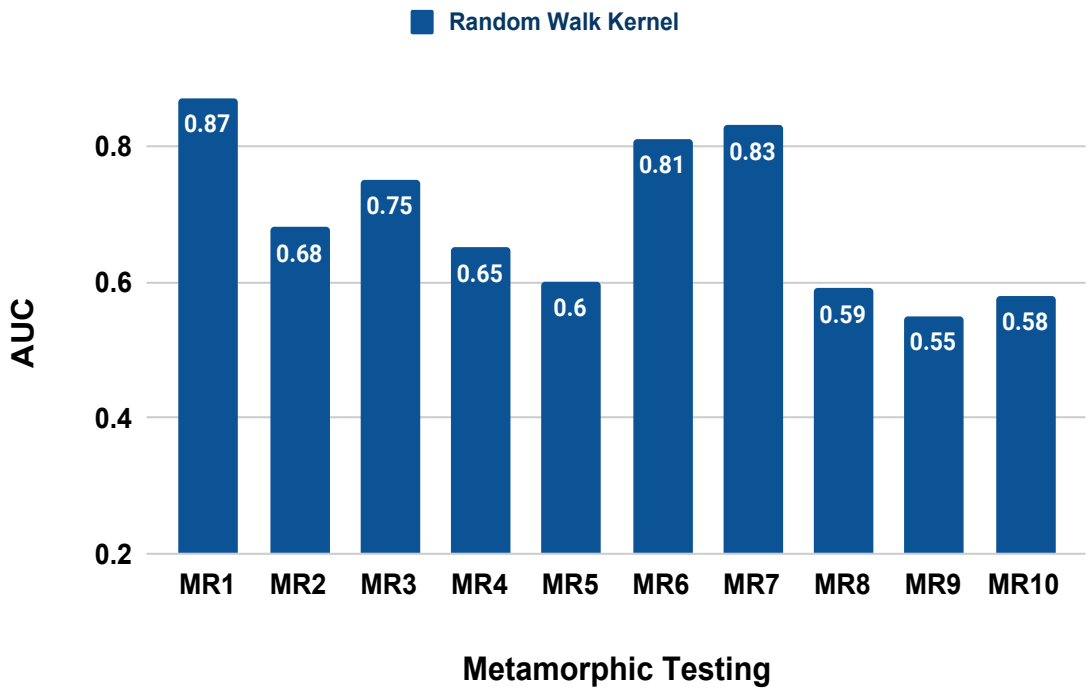

Figure 5.3: Prediction AUC score for each metamorphic relation on *random walk kernel* based SVM model with best parameters using the test set

Figure 5.3 shows the AUC scores for the test data set when the random walk

kernel is used with SVMs to train the predictive models. Among the 10 MRs, 4 MRs (i,e., MR1 (0.87), MR3 (0.75), MR6 (0.81), and MR7 (0.83)) show high AUC scores. The other MRs also reported AUC values higher than or equal to 0.55, indicating that this approach can differentiate between positive and negative instances for each MR. But further investigation is required to improve the prediction accuracy.

5.3.2 Performance of S3VMs

Metamorphic Testing	Control Flow Graph- Best λ
MR1	0.5
MR2	0.5
MR3	0.4
MR4	0.3
MR5	0.6
MR6	0.4
MR7	0.6
MR8	0.6
MR9	0.7
MR10	0.2

Table 5.2: Best λ for CFGs parameter value for each metamorphic relation for selecting the predictive model on validation set of 0.5 λ for CGs

The experiment is further extended by using the S3VMs to build the predictive model. Due to the long computation time for S3VMs, only two combinations of parameters are evaluated in this work. Firstly, λ values of 0.5, when computing random walk kernel using CGs of the subject function, is used with all the λ values when computing random walk kernel using CFGs of the method call function. Table

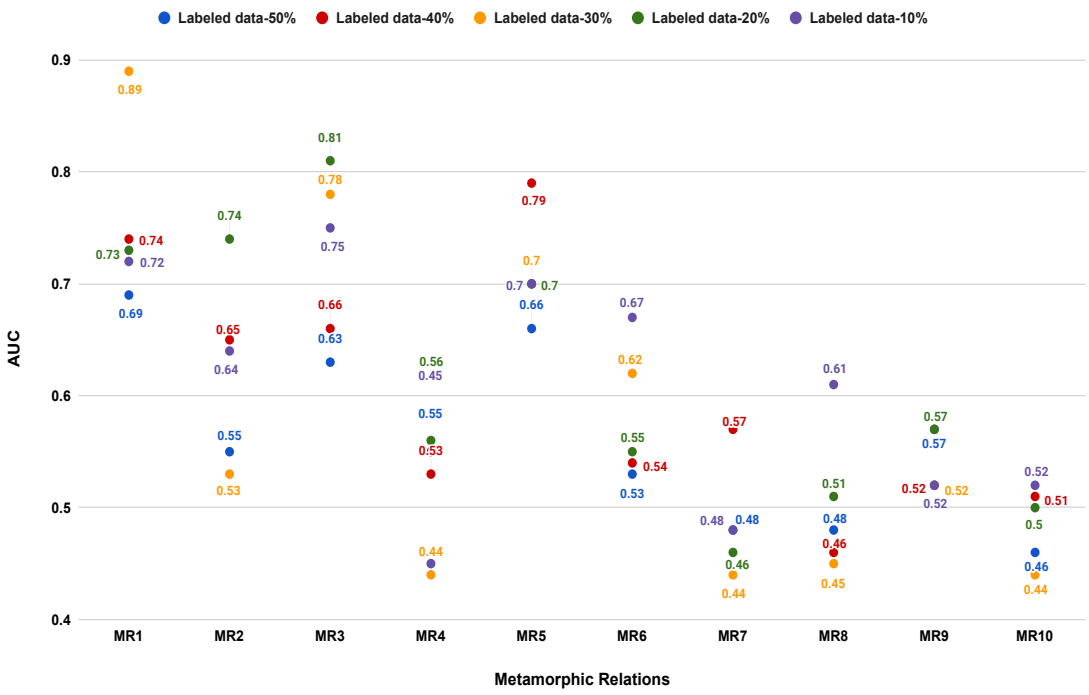

Figure 5.4: Prediction AUC score for *random walk kernel* based S3VM model with 0.5 λ parameters for CGs with 10%, 20%, 30%, 40% and 50% labeled data

5.2 lists the best λ for CFGs, for which the highest AUC values for each MR on the validation set are recorded. The best value for λ when using CFGs are different for different MRs. The most selected value recorded for the best λ is 0.6 when for CFGs for MR5, MR7, and MR8.

Figure 5.4 shows the AUC scores for the test data set when the random walk kernel and S3VM are used to train the predictive models. We can see that 11 points are higher or equal to 0.7, which are considered good AUC scores. With 10% labeled data, 3/10 MRs (i.e., MR1 (0.72), MR3 (0.75), and MR5(0.7)) show high AUC score. Among the 10 MRs, 4 MRs (i,e., MR1 (0.73), MR2 (0.74), MR3 (0.81), and MR5 (0.7)) show the high AUC scores with 20% labeled data. Also, when 30% of labeled data is used, again, 3/10 MRs (i.e., MR1 (0.89), MR3 (0.78), and MR5(0.7)) show a

high AUC score. Lastly, MR1 (0.74) and MR5 (0.79) show the high AUC score for 40% labeled data. According to the result, MR1, MR3, and MR5 have high accuracy scores across most of the ranges of labeled data. However, the ratio of positive and negative instances is the same for MR1 and MR3, but almost the opposite for MR5.

Secondly, λ values of 0.9, when computing random walk kernel using CGs of the subject function, is used with all the λ values when computing random walk kernel using CFGs of the method call function. Table 5.3 lists the best λ for CFGs, for which the highest AUC values for each MR on the validation set are recorded. The best value for λ for CFGs are different for different MRs. The most selected value recorded for the best λ is 0.8 when using CFGs for MR1, MR2, MR3, MR5, and MR8.

Figure 5.5 shows the AUC scores for the test data set when the random walk kernel and S3VMs are used to train the predictive models. We can see that 6 points are higher or equal to 0.7, which are considered good AUC scores. With 20% labeled data, 3/10 MRs (i.e., MR1 (0.71), MR3 (0.71), and MR5(0.88)) show the high AUC score. Also, the high score for MR5 (0.79) is observed when 30%, MR4 (0.75) is observed when 40% and MR5 (0.73) is observed when 50% of labeled data are used. Because of the long computation time, only two λ parameters for call graphs are used. According to the result, MR5 has high accuracy scores across the most of the ranges of labeled data. If all of them are considered, it might give better AUC scores. Therefore, further investigation is required to improve the prediction accuracy with this approach.

Metamorphic Testing	Control Flow Graph- Best λ
MR1	0.8
MR2	0.8
MR3	0.8
MR4	0.9
MR5	0.8
MR6	0.9
MR7	0.9
MR8	0.8
MR9	0.4
MR10	0.2

Table 5.3: Best λ for CFGs parameter value for each metamorphic relation for selecting the predictive model on validation set of 0.9 λ for CGs

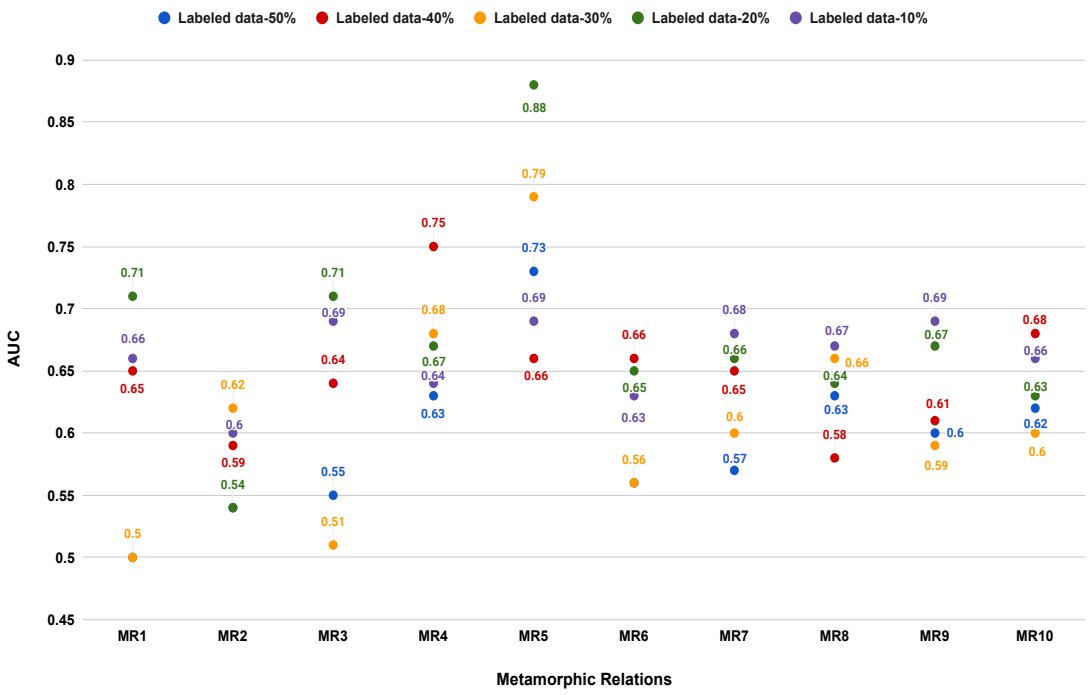

Figure 5.5: Prediction AUC score for *random walk kernel* based S3VM model with 0.9 λ parameters for CGs with 10%, 20%, 30%, 40% and 50% labeled data

CHAPTER SIX

CONCLUSIONS

The objective of this thesis is to evaluate the effectiveness of automatically predicting metamorphic relations using machine learning techniques. This chapter draws the overall conclusion of the thesis and the prospective future work in the focused area.

Metamorphic testing is a testing technique for programs that do not have a suitable test oracle. It is a mechanism of checking a set of properties called metamorphic relations between inputs and outputs. A violation of an MR indicates the presence of an error in the testing program. Manual identification of such MRs for different applications is difficult for the tester. In a previous work of Kanewala et al. [4], they have implemented this graph kernel-based machine learning method for predicting MRs using supervised learning and control flow graph features for simpler programs. Their results show that using a control flow graph feature to compute the graph kernel of a testing program for training a machine learning model is useful for predicting MRs. This thesis extends this work by evaluating the effectiveness of diverse graph representation and semi-supervised learning. Supervised and semi-supervised machine learning approaches are evaluated using control flow graph and call graph features, for programs that perform matrix calculations. Ten types of metamorphic relations are manually identified in this thesis for predicting MRs for matrix programs. Firstly, the MRs are predicted on a supervised model which is built using SVMs. After that, the supervised model is compared to a semi-supervised model that uses S3VMs to construct the predictive model, and both the random walk kernel and the graphlet kernel are used in both instances. Lastly, call graph information

about the functions are incorporated to calculate the random walk kernels to predict MRs, and the performance of the approach is evaluated. Three studies have shown promising results to adapt this approach for predicting MRs for scientific application.

Chapter 3 result shows that the random walk kernel performed better than the graphlet kernel for 7/10 MRs, and for the other MRs the AUC values are ≥ 0.74. In general, AUC ≥ 0.7 indicates that the trained model performs well. Therefore, graph kernels that are developed using walks in the graphs are highly effective in predicting metamorphic relations. So, it can be concluded that the proposed method using a supervised model can be considered as an effective model for predicting MRs for programs that perform matrix calculations.

Chapter 4 result shows that S3VMs performs better than SVMs when predicting 5/10 MRs. The effectiveness of the graph kernels using S3VMs is also evaluated, which shows once again that the random walk kernel is more effective when predicting 7/10 MRs. Also, a t-test is performed to determine the statistical significance of the improvement in the AUC scores. By looking at those results, it can be said that the addition of more unlabeled data points might increase accuracy scores. However, the addition of more unlabeled data is less expensive than adding labeled data for the improvement of the proposed approach. Therefore, the proposed method of using S3VMs can be an effective way to predict MRs.

Chapter 5 result shows that when CGs are used as the program representation, 4/10 MRs performed well when using SVMs to train the predictive model. Because of the long computation time, only two combinations of cases are shown. λ values of 0.5 and 0.9, when computing random walk kernel using CGs of the subject function, are used with all the λ values when computing random walk kernel using CFGs of the method call function. For those two cases, only 4/10 MRs, performed well with lower unlabelled data. If all possible cases are considered, it might give better AUC scores

for the MRs. However, the functions used in this experiment, as well as the functions of the method calls are comparatively small. Using functions with multiple lines of code (large functions) might also improve the AUC scores of the MRs. Therefore, machine learning algorithms, together with updated random walk kernel methods that are calculated by incorporating call graph information of the functions, can be used for predicting metamorphic relations for a previously unseen program with further modifications.

From the results of Chapter 3, we can conclude that MR2, MR5, and MR7 are performing well for the graph kernel-based supervised machine learning method. All of them have an AUC score higher than or equal to 0.9 when using a random walk kernel. In Chapter 4 , it can be observed that MR1, MR2, MR3, and MR5 are predicted with high AUC across the different ranges of labeled data when using random walk kernel for the S3VM model, and also MR1, MR2, MR3, MR4, and MR5 are predicted with high AUC across the different ranges of labeled data when using graphlet kernel for the S3VM model. We can conclude that MR1, MR2, MR3, and MR5 are performing well when semi-supervised machine learning is used to build a predictive model. According to Chapter 5, MR1, MR6, and MR7 are performing well when call graphs are used to compute the random walk kernel, which is then used to build the supervised predictive model. All of them have an AUC score higher than 0.8. The results show that MR1, MR3, and MR5 have high accuracy scores across most of the ranges of labeled data when 0.5 λ is used for the S3VM model. Also, MR5 has high accuracy scores across most of the ranges of labeled data when 0.9 λ is used for the S3VM model.

Another observation is that MR8, MR9, and MR10 performed poorly across all the methods of this study. These 3 MRs have high positive and low negative instances and the ratio is about the same for all. Most of them have an AUC score

lower than 0.7. Moreover, we can see that MR7 performs well across the methods using supervised models, and MR5 performs well across the methods using semi-supervised models. However, MR7 and MR5 have low positive and high negative instances. So, it can be concluded that the low positive instances does not have an effect on the prediction accuracy of the MRs.

6.1 Threat to Validity

There are some possible threats when it comes to validating the graph kernel-based machine learning method examined in this study. Two types of validity threats are discussed here, the external threat and the internal threat. One of the main threats for the validation is the generalization of the results of the study for other cases. The study uses 93 mathematical functions that perform matrix calculations, which is a comparatively small data set. Therefore, the obtained results cannot explicitly confirm that this approach will scale to industrial-sized software.

The method uses a few third-party tools, which might contain potential faults and can be a threat to the proposed method. This type of threat can occur in terms of internal validity. The study used scikit-learn, Soot, and QN-S3VM to produce the results for the experiments. Many researchers in the computer science community use Scikit-learn and Soot for their work as they are well-tested and have limited faults. On the other hand, QN-S3VM is not highly used and is more likely to contain errors than scikit-learn or Soot.

6.2 Future Work

The proposed method can be extended in several directions. Firstly, to avoid the threat to external validity, a more extensive data set with more functions can be used,

which can lead to better accuracy. The proposed method is built to predict single metamorphic relation, which means a binary classification based approach is used to train the predictive models. In this study, most of the functions have more than one metamorphic relations. This issue can be addressed by modifying the method into a multi-class problem. With the recent popularity of deep learning, we intend to utilize neural networks as the underlying machine learning model, which will also negate the need for hand-engineering features.

Furthermore, the functions used in this research take matrices as inputs, whereas real-life programs can take different forms of inputs such as volumes, and graphs, etc. Therefore, distinct machine learning models can be trained for different kinds of programs, or a general model can be trained using a wider variety of data set. Also, the method can be modified to extend the MR prediction scope beyond the function level. However, the proposed method can be modified to predict MRs for other programming languages such as Python, C, and Fortran, which can be beneficial for the scientific community. Other than the graph kernel-based approach, a text classification method can be used to predict MRs for functions using their documentation.

REFERENCES CITED

REFERENCES CITED

[1] R. Sanders and D. Kelly, "The challenge of testing scientific software," 01 2008.

[2] L. Hardesty, "Explained: Matrices," Dec 2013.

[3] E. J. Weyuker, "On Testing Non-Testable Programs," *The Computer Journal*, vol. 25, pp. 465–470, 11 1982.

[4] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels," *Softw. Test. Verif. Reliab.*, vol. 26, p. 245–269, May 2016.

[5] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," 01 1998.

[6] V. H. S. Durelli, R. S. Durelli, S. S. O. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. de Paiva Guimarães, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, vol. 68, pp. 1189–1212, 2019.

[7] S. Segura, G. Fraser, A. B. Sánchez, and A. R. Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, pp. 805–824, 2016.

[8] T. Y. Chen, T. H. Tse, and Z. Zhou, "Fault-based testing without the need of oracles," *Information and Software Technology*, vol. 45, no. 1, pp. 1 – 9, 2003.

[9] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 1–10, Nov 2013.

[10] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Wiley Publishing, 3rd ed., 2011.

[11] P. Ammann and J. Offutt, *Introduction to Software Testing*. USA: Cambridge University Press, 1 ed., 2008.

[12] S. Vergilio, J. Maldonado, and M. Jino, "Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction.," *J. Braz. Comp. Soc.*, vol. 12, pp. 71–86, 02 2006.

[13] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, pp. 507–525, 2015.

[14] W. E. Howden, "Theoretical and empirical studies of program testing," in *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, p. 305–311, IEEE Press, 1978.

[15] R. Sanders and D. Kelly, "The challenge of testing scientific software," 01 2008.

[16] C. Murphy, G. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing.," pp. 867–872, 01 2008.

[17] P. Louridas and C. Ebert, "Machine learning," *IEEE Software*, vol. 33, pp. 110–115, 09 2016.

[18] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. The MIT Press, 2012.

[19] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. USA: Cambridge University Press, 2014.

[20] H. Liu, X. Liu, and T. Y. Chen, "A new method for constructing metamorphic relations," in *2012 12th International Conference on Quality Software*, pp. 59–68, Aug 2012.

[21] G. Dong, B. Xu, L. Chen, C. Nie, and L. Wang, "Case studies on testing with compositional metamorphic relations," vol. 24, pp. 437–443, 12 2008.

[22] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), pp. 701–712, ACM, 2014.

[23] F. H. Su, J. Bell, C. Murphy, and G. Kaiser, "Dynamic inference of likely metamorphic properties to support differential testing," in *Proceedings of the 10th International Workshop on Automation of Software Test*, AST '15, (Piscataway, NJ, USA), pp. 55–59, IEEE Press, 2015.

[24] T. Y. Chen, P. L. Poon, S. F. Tang, and T. H. Tse, "Dessert: a divide-and-conquer methodology for identifying categories, choices, and choice relations for test case generation," *IEEE Transactions on Software Engineering*, vol. 38, pp. 794–809, July 2012.

[25] T. Y. Chen, P. L. Poon, and X. Xie, "Metric: Metamorphic relation identification based on the category-choice framework," *The Journal of systems and software*, vol. 116, pp. 177–190, 2016.

[26] B. Hardin and U. Kanewala, "Using semi-supervised learning for predicting metamorphic relations," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, MET '18, (New York, NY, USA), p. 14–17, Association for Computing Machinery, 2018.

[27] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, (New York, NY, USA), pp. 1–19, ACM, 1970.

[28] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[29] T. Hofmann, B. Schölkopf, and A. Smola, "Kernel methods in machine learning," *The Annals of Statistics*, vol. 36, 01 2007.

[30] R. Sharan and T. Ideker, "Modeling cellular machinery through biological network comparison," *Nature Biotechnology*, vol. 24, pp. 427–433, 2006.

[31] H. Hosoya, *Introduction to Graph Theory*, pp. 1–36. Dordrecht: Springer Netherlands, 1994.

[32] R. Kumar, J. Novak, and A. Tomkins, "Structure and evolution of online social networks," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, (New York, NY, USA), p. 611–617, Association for Computing Machinery, 2006.

[33] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," vol. 129-143, pp. 129–143, 01 2003.

[34] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," vol. 129-143, pp. 129–143, 01 2003.

[35] K. Borgwardt, underlineCS, S. Schönauer, S. Vishwanathan, A. Smola, and H. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21 Suppl 1, pp. i47–56, 01 2005.

[36] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods*. USA: Cambridge University Press, 1999.

[37] G. Wiederschain, "Data mining techniques for the life sciences (olivero carugo and frank eisenhaber (eds.), in series "springer protocols. methods in molecular biology", vol. 609, humana press, 2010, 407 p., 110)," *Biochemistry (Moscow)*, vol. 76, 04 2011.

[38] D. J. Hand and C. Anagnostopoulos, "When is the area under the receiver operating characteristic curve an appropriate measure of classifier performance?," *Pattern Recognition Letters*, vol. 34, no. 5, pp. 492–495, 2013.

[39] K. P. Bennett and A. Demiriz, "Semi-supervised support vector machines," in *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, (Cambridge, MA, USA), p. 368–374, MIT Press, 1999.

[40] F. Gieseke, A. Airola, T. Pahikkala, and O. Kramer, "Fast and simple gradient-based optimization for semi-supervised support vector machines," *Neurocomput.*, vol. 123, p. 23–32, Jan. 2014.

APPENDIX

APPENDIX: NAME OF THE SUBJECT FUNCTIONS

Function name	**Open source project**
divide(double)	la4j (Linear Algebra for Java)
shuffle()	la4j (Linear Algebra for Java)
sliceBottomRight(intint)	la4j (Linear Algebra for Java)
rotate()	la4j (Linear Algebra for Java)
removeRow(int)	la4j (Linear Algebra for Java)
insert(orgla4jMatrixintintintintintint)	la4j (Linear Algebra for Java)
power(int)	la4j (Linear Algebra for Java)
add(double)	la4j (Linear Algebra for Java)
insertColumn(intorgla4jVector)	la4j (Linear Algebra for Java)
sliceTopLeft(intint)	la4j (Linear Algebra for Java)
select(int[]int[])	la4j (Linear Algebra for Java)
insert(orgla4jMatrixintint)	la4j (Linear Algebra for Java)
insertRow(intorgla4jVector)	la4j (Linear Algebra for Java)
multiply(orgla4jMatrix)	la4j (Linear Algebra for Java)
slice(intintintint)	la4j (Linear Algebra for Java)
insert(orgla4jMatrix)	la4j (Linear Algebra for Java)
removeColumn(int)	la4j (Linear Algebra for Java)
add(orgla4jMatrix)	la4j (Linear Algebra for Java)
transpose()	la4j (Linear Algebra for Java)
multiply(double)	la4j (Linear Algebra for Java)
insert(orgla4jMatrixintintintint)	la4j (Linear Algebra for Java)
equals(orgla4jMatrixdouble)	la4j (Linear Algebra for Java)
swapRows(intint)	la4j (Linear Algebra for Java)
setColumn(intorgla4jVector)	la4j (Linear Algebra for Java)
setColumn(intdouble)	la4j (Linear Algebra for Java)
swapColumns(intint)	la4j (Linear Algebra for Java)
setRow(intorgla4jVector)	la4j (Linear Algebra for Java)
setRow(intdouble)	la4j (Linear Algebra for Java)
times(double)	JAMA (Java Matrix package)
minus(JamaMatrix)	JAMA (Java Matrix package)
ArrayLeftDivideEquals(JamaMatrix)	JAMA (Java Matrix package)
plus(JamaMatrix)	JAMA (Java Matrix package)
timesEquals(double)	JAMA (Java Matrix package)
uminus()	JAMA (Java Matrix package)
times(JamaMatrix)	JAMA (Java Matrix package)
getMatrix(int[]int[])	JAMA (Java Matrix package)

minusEquals(JamaMatrix)	JAMA (Java Matrix package)
ArrayTimes(JamaMatrix)	JAMA (Java Matrix package)
ArrayLeftDivide(JamaMatrix)	JAMA (Java Matrix package)
inverse()	JAMA (Java Matrix package)
plusEquals(JamaMatrix)	JAMA (Java Matrix package)
ArrayTimesEquals(JamaMatrix)	JAMA (Java Matrix package)
ArrayRightDivideEquals(JamaMatrix)	JAMA (Java Matrix package)
transpose()	JAMA (Java Matrix package)
getMatrix(intintint[])	JAMA (Java Matrix package)
getMatrix(int[]intint)	JAMA (Java Matrix package)
getMatrix(intintintint)	JAMA (Java Matrix package)
ArrayRightDivide(JamaMatrix)	JAMA (Java Matrix package)
setMatrix(intintint[]JamaMatrix)	JAMA (Java Matrix package)
setMatrix(intintintintJamaMatrix)	JAMA (Java Matrix package)
setMatrix(int[]intintJamaMatrix)	JAMA (Java Matrix package)
setMatrix(int[]int[]JamaMatrix)	JAMA (Java Matrix package)
set(intintdouble)	JAMA (Java Matrix package)
add(org.apache.commons.math3. linear.Array2DRowRealMatrix)	Apache Commons Math Library
multiply(org.apache.commons. math3.linear.Array2DRowRealMatrix)	Apache Commons Math Library
subtract(org.apache.commons. math3.linear.Array2DRowRealMatrix)	Apache Commons Math Library
createMatrix(org.apache.commons. math3.linear.Array2DRowRealMatrix)	Apache Commons Math Library
setEntry(org.apache.commons. math3.linear.Array2DRowRealMatrix)	Apache Commons Math Library
setSubMatrix(org.apache.commons. math3.linear.Array2DRowRealMatrix)	Apache Commons Math Library
addToEntry(org.apache.commons.math3. linear.Array2DRowRealMatrix)	Apache Commons Math Library
multiplyEntry(org.apache.commons.math3. linear.Array2DRowRealMatrix)	Apache Commons Math Library
subtract(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
multiply(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
getColumnMatrix(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
scalarMultiply(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library

add(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
getRowMatrix(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
power(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
scalarAdd(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
equals(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
isSquare(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
setRowVector(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
setRowMatrix(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
setRow(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
setColumnVector(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
setColumnMatrix(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
copySubMatrix(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
setColumn(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
setSubMatrix(org.apache.commons.math3. linear.AbstractRealMatrix)	Apache Commons Math Library
scalarMultiply(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
setSubMatrix(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
addToEntry(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
setColumn(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
multiplyEntry(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
setColumnMatrix(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library

setRow(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
setEntry(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
setRowMatrix(org.apache.commons.math3. linear.BlockRealMatrix)	Apache Commons Math Library
multiply(org.apache.commons.math3. linear.OpenMapRealMatrix)	Apache Commons Math Library
add(org.apache.commons.math3. linear.OpenMapRealMatrix)	Apache Commons Math Library
multiplyEntry(org.apache.commons.math3. linear.OpenMapRealMatrix)	Apache Commons Math Library
addToEntry(org.apache.commons.math3. linear.OpenMapRealMatrix)	Apache Commons Math Library
inverse(org.apache.commons.math3. linear.DiagonalMatrix)	Apache Commons Math Library

Table A.1: Code base used in the experiment