Dissertation

TESTING SCIENTIFIC SOFTWARE: TECHNIQUES FOR AUTOMATIC DETECTION OF METAMORPHIC RELATIONS

Submitted by

Upulee G. Kanewala

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2015

Doctoral Committee:

Advisor: James M. Bieman

Sudipto Ghosh
Chuck Anderson
Jay Breidt

Abstract

TESTING SCIENTIFIC SOFTWARE: TECHNIQUES FOR AUTOMATIC DETECTION OF METAMORPHIC RELATIONS

Scientific software plays an important role in critical decision making in fields such as the nuclear industry, medicine, and the military. Systematic testing of such software can help to ensure that it works as expected. Comprehensive, automated software testing requires an oracle to check whether the output produced by a test case matches the expected behavior of the program. But the challenges in creating suitable oracles limit the ability to perform automated testing of scientific software.

For some programs, creating an oracle may be not possible since the correct output is not known a priori. Further, it may be impractical to implement an oracle for an arbitrary input due to the complexity of a program. The software testing community refers to such programs as *non-testable*. Many scientific programs fall into this category of non-testable programs, since they are either written to find answers that are previously unknown or they perform complex calculations. In this work, we developed techniques to automatically predict *metamorphic relations* by analyzing the program structure. These metamorphic relations can serve as automated partial test oracles in scientific software.

Metamorphic testing is a method for automating the testing process for programs without test oracles. This technique operates by checking whether a program behaves according to a certain set of properties called *metamorphic relations*. A metamorphic relation is a relationship between multiple input and output pairs of the program. It specifies how the output should change following a specific change made to the input. A change in the output

that differs from what is specified by the metamorphic relation indicates a fault in the program. Metamorphic testing can be effective in testing machine learning applications, bioinformatics programs, health-care simulations, partial differential equations and other programs.

Unfortunately, finding appropriate *metamorphic relations* for use in metamorphic testing remains a labor intensive task that is generally performed by a domain expert or a programmer. In this work we applied novel machine learning based approaches to automatically derive metamorphic relations.

We first evaluated the effectiveness of modeling the metamorphic relation prediction problem as a binary classification problem. We found that support vector machines are the most effective binary classifiers for predicting metamorphic relations. We also found that using walk-based graph kernels for feature extraction from graph-based program representations further improves the prediction accuracy. In addition, incorporating mathematical properties of operations in the graph kernel computation improves the prediction accuracy. Further, we found that control flow information of a function are more effective than data dependency information for predicting metamorphic relations. Finally we investigated the possibility of creating multi-label classifiers that can predict multiple metamorphic relations using a single classifier. Our empirical studies show that multi-label classifiers are not effective as binary classifiers for predicting metamorphic relations.

Automated testing will make the testing process faster, reduce the testing cost and make it more reliable. Automated testing requires automated test oracles. Automatically discovering metamorphic relations is an important step towards automating oracle creation.

Work presented here is the first attempt towards developing automated techniques for deriving metamorphic relations. Our work contributes toward automating the testing process of programs that face oracle problems.

Acknowledgements

First, I would like to convey my gratitude to my adviser, Dr. James M. Bieman, for providing me invaluable guidance in my research as well as publishing. I am especially thankful to him for giving me the freedom to choose my own research direction.

I would like to thank my committee members Dr. Sudipto Ghosh, Dr. Chuck Anderson, and Dr. Jay Breidt for providing valuable advice throughout different stages of my research. I would also like to thank Dr. Asa Ben-Hur for sharing his insights and knowledge about machine learning.

The work described in this dissertation was supported by the National Institute Of General Medical Sciences (award number 1R01GM096192). I would also like tho thank all the members to the SAXS group at Colorado State University. I am thankful to the Information Science and Technology Center (ISTeC) at Colorado State University and the National Science Foundation (NSF) for providing me with financial support.

I would like to thank my parents, K. Harischandra and R.M.U.K Ranathunga, my two younger sisters Gayanee Kanewala, and Udaree Kanewala for always being there for me throughout my academic life. Last but not least, I thank my husband, Indika Kahanda for his continuous support and encouragement in my research work.

Dedication

Dedicated to my loving parents and my loving husband.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Scientific software is widely used in science and engineering. For example, in nuclear weapons simulations, code is used to determine the impact of modifications, since these weapons cannot be field tested [1]. Climate models make climate predictions and assess climate change [2]. In addition, results obtained from such software are used as evidence for research publications [3]. Due to the lack of systematic testing of scientific software, faults can remain undetected. Some of these faults can cause incorrect program outputs without causing a program to crash. Software faults such as one-off errors have caused loss of precision in seismic data processing programs [4]. Other software faults have compromised coordinate measuring machine (CMM) performance [5]. In addition, there have been several retractions of published work due to software faults [6]. Hatton et al. [7] found that several software systems written for geoscientists produced reasonable yet essentially different results. There were situations where scientists believed that they needed to modify a physics model or develop new algorithms but later discovered that the real problem was small faults in their code [8]. Comprehensive automated testing of scientific software can help to ensure that it is implemented correctly.

One of the greatest challenges in software testing is the oracle problem. Automated testing requires automated test oracles, but such oracles may not exist. This problem commonly arises when testing scientific software. Many scientific applications fall into the category of "non-testable programs" [9] where an oracle is unavailable or too difficult to implement. In such situations, a domain expert must manually check that the output produced by an application is correct for a selected set of inputs. Further, Sanders et al. [10] found that,

due to a lack of background knowledge in software engineering, scientists tend to conduct testing in an unsystematic way. This situation makes it difficult for testing to detect subtle faults such as one-off errors, and hinders the automation of the testing process. A recent survey conducted by Joppa et al. showed that, when adopting scientific software, only 8% of the scientists independently validate the software and the others choose to use the software simply because it was published in a peer-reviewed journal or because of personal opinions and recommendations [11]. Therefore undetected subtle faults can affect findings in multiple studies that use the same scientific software. Techniques that can make it easier to test software without oracles are clearly needed [12].

Metamorphic testing is a technique, introduced by Chen et al. [13], that can be used to test programs that do not have oracles. This technique operates by checking whether a program under test behaves according to an expected set of properties known as *metamorphic relations*. A *metamorphic relation* specifies how a particular change to the input of the program should change the output [14]. For example, in a program that calculates the average of an integer array, randomly permuting the order of the elements in the input array should not change the result. This property can be used as a metamorphic relation to test this program.

Violation of a metamorphic relation occurs when the change in the output differs from what is specified by the considered metamorphic relation. Satisfying a particular metamorphic relation does not guarantee that the program is implemented correctly. However, a violation of a metamorphic relation indicates that the program contains faults. Previous studies show that metamorphic testing can be an effective way to test programs without oracles [14, 15].

Enumerating a set of metamorphic relations that should be satisfied by a program is a critical initial task in applying metamorphic testing [16, 17]. Currently, a tester or a developer has to manually identify metamorphic relations using her knowledge of the program under test; this manual process can easily miss important metamorphic relations that could reveal faults.

In this dissertation we develop novel techniques for automatically deriving metamorphic relations for a given program. As described in Chapter 3, we use machine learning techniques to develop predictive models to find metamorphic relations. **Initially we modeled the metamorphic relation prediction problem as a binary classification problem.** We developed a set of features that represents information about nodes and paths in a function's (or method's) control flow graph. Then we used these features to build a predictive model using machine learning techniques to classify whether a function exhibits a particular metamorphic relation or not. We evaluated the effectiveness of three machine learning classification algorithms for predicting metamorphic relations using a set of numerical functions. Our results showed that support vector machines were the most effective machine learning algorithm for predicting metamorphic relations.

Chapter 4 describes how **we applied graph kernels to improve the effectiveness of feature extraction from graph-based function representations.** A graph kernel calculates a similarity score for a pair of graphs using different graph substructures. We developed two graph kernels that can be applied to graph-based function representations. These two graph kernels use walks and subgraphs as graph substructures when computing the kernels. We found that graph kernels computed using walks are more effective than the graph kernels computed using subgraphs for predicting metamorphic relations. We

also found that utilizing properties of mathematical operations when computing the graph kernels will further improve the predictive effectiveness of the classifiers. **Further, we evaluated the prediction effectiveness of different types of semantic information of a function.** For this we used control flow and data dependency information about a function. Results of our empirical studies show that control flow information is more effective than data dependency information for predicting metamorphic relations. But, for some metamorphic relations, using both types of information to create the predictive models improved the prediction accuracy.

Finally, **we evaluated the effectiveness of modeling the metamorphic relation prediction problem as a multi-label classification problem** (Chapter 5). Multi-label classifiers assign multiple labels to each instance. Instead of training separate binary classifiers for each metamorphic relation as above, we investigated the effectiveness of developing a single multi-label classifier to predict multiple metamorphic relations at once. Our empirical studies showed that multi-label classifiers are not effective as binary classifiers for predicting metamorphic relations.

We utilized this novel approach to test a scientific program that was developed in-house to analyze small angle x-ray scattering (SAXS) data (Chapter 4). We were able to detect 90% of artificially inserted faults using the predicted metamorphic relations derived using our approach.

The main contribution of our work is the development of techniques to automatically create oracles for automated testing of scientific software. Automatically predicted metamorphic relations will serve as test oracles that determine if a test case passes or fails. Our work will support automation of the metamorphic testing process without requiring domain

experts to manually derive metamorphic relations. In addition, our work utilized advanced machine learning algorithms for the task of predicting metamorphic relations. To the best of our knowledge, this is the first time that graph kernels and multi-label classifiers have been incorporated to solve an important problem in the field of software engineering. Finally, our work is the first attempt at developing techniques to automatically infer metamorphic relations. The ability to automatically create test oracles is an important advancement in automated software testing, which will help to reduce the human labor involved in the testing process.

CHAPTER 2

Background

This chapter provides background and context for the work presented in this dissertation. We present the motivation for this work through some real-world examples that we encountered along with information found through a systematic literature survey [12] in Section 2.1. Then we describe the metamorphic testing technique in Section 2.2. Finally, we discuss the machine learning techniques used in this work in Section 2.3.

2.1. Oracle problem in scientific software testing

This research began when we were asked to test a program written to analyze small angle x-ray scattering data (SAXS)[1]. As with many scientific programs the SAXS program fell into the category of *non-testable programs*. The absence of a test oracles made it difficult to conduct systematic testing on the SAXS program, since we needed a domain expert to manually evaluate the test results.

Through a systematic literature survey [12] we conducted, we found that oracle problems are common in testing scientific software. More than 30% of the primary studies identified in the systematic literature survey reported the lack of an oracle as a serious problem [12]. The following characteristics of scientific software make it challenging to create an oracle that is required to perform systematic testing:

(1) Some scientific software is written to find answers that are previously unknown. Therefore only approximate solutions might be available [18, 19, 9, 20, 21].

(2) It is difficult to determine the correct output for software written to test scientific theory that involves complex calculations or simulations. Further, some programs

[1] `http://www.cs.colostate.edu/hpc/SAXS/index.php`

produce complex outputs making it difficult to determine the expected output [22, 10, 23–27, 9, 28].

(3) Due to the inherent uncertainties in models, some scientific programs do not give a single correct answer for a given set of inputs. This makes determining the expected behavior of the software a difficult task, which may depend on a domain expert's opinion [5].

(4) Requirements are unclear or uncertain up-front due to the exploratory nature of the software. Therefore developing oracles based on requirements is not commonly done [22, 29, 27, 30].

(5) Choosing suitable tolerances for an oracle when testing numerical programs is difficult due to the involvement of complex floating point computations [26, 31–33].

We also found several methods used by scientific software developers to overcome oracle problems:

(1) A *pseudo oracle* is an independently developed program that fulfills the same specification as the program under test [5, 29, 34, 35, 1, 10, 9, 36, 4]. For example, Murphy *et al.* used pseudo oracles for testing a machine learning algorithm [23].

Limitations: A pseudo oracle may not include some special features/treatments available in the program under test and it is difficult to decide whether the oracle or the program is faulty when the answers do not agree [37]. Pseudo oracles make the assumption that independently developed reference models will not result in the same failures. But Brilliant *et al.* found that even independently developed programs might produce the same failures [38].

(2) Solutions obtained analytically can serve as oracles. Using analytical solutions is sometimes preferred over pseudo oracles since they can identify common algorithmic errors among the implementations. For example, a theoretically calculated rate of convergence can be compared with the rate produced by the code to check for faults in the program [5, 25, 34].

Limitations: Analytical solutions may not be available for every application [37] and may not be accurate due to human errors [10].

(3) Experimentally obtained results can be used as oracles [5, 25, 29, 1, 10, 39].

Limitations: It is difficult to determine whether an error is due to a fault in the code or due to an error made during the model creation [37]. In some situations experiments cannot be conducted due to high cost, legal or safety issues [20].

(4) Measurements values obtained from natural events can be used as oracles.

Limitations: Measurements may not be accurate and are usually limited due to the high cost or danger involved in obtaining them [25, 3].

(5) Using the professional judgment of scientists [3, 32, 40, 10]

Limitations: Scientists can miss faults due to misinterpretations and lack of data. In addition, some faults can produce small changes in the output that might be difficult to identify [40]. Further, the scientist may not provide objective judgments [10].

(6) Using simplified data so that the correctness can be determined easily [9].

Limitations: It is not sufficient to test using only simple data; simple test cases may not uncover faults such as round-off problems, truncation errors, overflow conditions, etc [41]. Further such tests do not represent how the code is actually used [10].

(7) Statistical oracle: verifies statistical characteristics of test results [42].

Limitations: Decisions by a statistical oracle may not always be correct. Further a statistical oracle cannot decide whether a single test case has passed or failed [42].

(8) Reference data sets: Cox *et al.* created reference data sets based on the functional specification of the program that can be used for black-box testing of scientific programs [43].

Limitations: When using reference data sets, it is difficult to determine whether the error is due to using unsuitable equations or due to a fault in the code.

Using these oracles, scientific software developers often only test whether a program produces obvious failures such as crashes or infeasible outputs. Therefore developing techniques to create oracles that can be used for systematic testing of these programs will be highly useful for improving the quality of scientific software.

2.2. Metamorphic testing (MT)

Metamorphic testing was originally presented as a technique to create additional test cases using existing test cases, especially the tests that did not result in any failures [13]. Soon it was found that MT can be used to test programs that do not have test oracles [14, 15]. Metamorphic testing supports the creation of follow-up test cases from existing test cases as follows:

(1) Identify an appropriate set of metamorphic relations that the program under test should satisfy.

(2) Create a set of *initial* test cases using techniques such as random testing, structural testing or fault based testing.

```
public static int addValues(int a[]){
        int sum=0;
        for(int i=0;i<a.length;i++){
                sum+=a[i];}
        return sum;}
```

FIGURE 2.1. Function for calculating the sum of elements in an array

(3) Create *follow-up* test cases by applying the input transformations required by the identified metamorphic relations in step 1 to each initial test case.

(4) Execute the initial and follow-up test case pairs to check whether the output change complies with the change predicted by the metamorphic relation. A runtime violation of a metamorphic relation during testing indicates a fault or faults in the program under test.

Since metamorphic testing checks the relationship between inputs and outputs of multiple executions of the program under test, this method can be used when the correct result of individual executions are not known.

Consider the function in Figure 2.1 that calculates the sum of integers in an array a. Randomly permuting the order of the elements in a should not change the result. This is the permutative metamorphic relation in Table 3.3. Further, adding a positive integer k to every element in a should increase the result by $k \times length(a)$. This is the additive metamorphic relation in Table 3.3. Therefore, using these two relations, two follow-up test cases can be created for every initial test case and the outputs of the follow-up test cases can be predicted using the initial test case output.

2.2.1. PREVIOUS WORK ON MT. Metamorphic testing has been used to test applications without oracles in different areas. Xie et al. [16] used metamorphic testing to test machine learning applications. Metamorphic testing was used to test simulation software such as

health care simulations [19] and Monte Carlo modeling [44]. Metamorphic testing has been used effectively in bioinformatics [24], computer graphics [45] and for testing programs with partial differential equations [46]. Murphy et al. [47] show how to automatically convert a set of metamorphic relations for a function into appropriate test cases and check whether the metamorphic relations hold when the program is executed. However they specify the metamorphic relations manually.

Metamorphic testing has also been used to test programs at the system level. Murphy et al. developed a method for automating system level metamorphic testing [48]. In this work, they also describe a method called *heuristic metamorphic testing* for testing applications with non-determinism. All of these approaches can benefit from our method for automatically finding likely metamorphic relations.

Our work contributes to this body of knowledge by investigating approaches to automatically derive metamorphic relations for a given program, which have not been investigated previously. Our work demonstrates the possibility of automatically predicting metamorphic relations from a function's source code. This will help to automate the MT process starting by deriving MRs, thus increase the applicability of MT.

2.3. Machine Learning

Machine learning methods focus on providing computer programs the ability to make better decisions based on experience [49]. Usually, the set of examples used by a machine learning algorithm are divided into two subsets: a *training set* and a *test set*. The *training set* is used to create the predictive model, while the *test set* is used to evaluate the performance of the predictive model.

Supervised learning is one machine learning method, where a set of labeled examples is used to learn a target function. The target function maps the input to a desired set of outputs (labels). Input to a supervised classification algorithm is a set of training data $S = \{s_1, s_2, ..., s_n\}$. Each *training instance* $s_i \in S$ takes the form $\{x_1, x_2, ..., x_m, c_i\}$, where x_j is a *feature* and c_i is the class label of the training instance s_i. A *feature* is a measurable property of an instance.

In this work, we model metamorphic relation prediction as a supervised learning problem. For a given metamorphic relation we create a supervised classification model using features extracted from a set of functions already known to satisfy/not satisfy the considered metamorphic relation. Then the trained classification model is used to predict whether a previously unseen function should satisfy the considered metamorphic relation or not.

2.3.1. Binary classification. In binary classification the class label can take only one of two possible values $(+1/-1)$. Input to a binary classification algorithm is a set of training examples $E = \{\mathbf{x}_i, c_i | \mathbf{x}_i \in \mathcal{X}, c_i = 1 \text{ or } -1\}$. Each $\mathbf{x}_i = x_1, x_2, ..., x_m$ is called a *training instance*, where x_j is a *feature* and c_i is the class label of the training instance. We use three binary classification algorithms in our work: *Decision Trees* [50], *Support Vector Machines (SVMs)* [51], and *Artificial Neural Networks* [52].

2.3.1.1. *Decision Trees (DT).* In decision tree learning, the target function is a decision tree. In classification, a decision tree maps the input to a binary label. Internal nodes of a decision tree test a feature in the input and leaf nodes assign a label. We used the J48 Java implementation of the C4.5 [53] decision tree generation algorithm from the WEKA [54] tool kit. When choosing a feature for an internal node, the C4.5 algorithm chooses the feature with the highest information gain [55].

2.3.1.2. *Support Vector Machines (SVMs).* SVM [51] is another supervised learning algorithm that is used for classification. SVM creates a hyper-plane in a high dimensional space that can separate the instances in the training set according to their class labels. When a linear separation cannot be found in the original feature space, SVMs use *kernel functions* to map the training data into a higher dimensional feature space. Then the SVM creates a linear separator in this higher dimensional feature space, which can be used to classify unseen data instances. In this work we used the SVM implementation in the PyML Toolkit[2].

2.3.1.3. *Artificial Neural Networks (ANNs).* ANN is a learning algorithm inspired by biological neural networks [52]. ANN is a made up of multiple simple processing units called *neurons*. These neurons perform a computation within itself using its input and produce an output. A neuron typically uses a non-linear activation function to produce the output. Neurons are connected through weighted connections, which represents knowledge in an ANN. Typically neurons are organized as layers in an ANN. The features are supplied to the *input layer*. This information is transferred to one or more hidden layers, where the actual processing is done. Hidden layers are linked to an *output layer* that produces the output. In this work used the ANN implementations developed by Dr. Chuck Anderson[3].

2.3.2. Multi-label Classification. In multi-label classification each example is associated with multiple labels. The goal is to learn a predictive model from the training data that will select a set of relevant labels for a previously unseen example [56].

Input to a multi-label classification algorithm is a set of training examples $E = \{\mathbf{x}_i, \mathcal{Y}_i | \mathbf{x}_i \in \mathcal{X}, \mathcal{Y}_i \in \mathcal{L}, 1 \leq i \leq N\}$. Let $\mathcal{X}$ be the example space consisting of tuples of $\mathbf{x}_i = (x_{i_1}, x_{i_2}, ..., x_{i_D})$. D is the number of features in a training example. Let $\mathcal{L} = \{\lambda_1, \lambda_1, ..., \lambda_Q\}$

[2]http://pyml.sourceforge.net/
[3]http://www.cs.colostate.edu/ anderson/cs645notebooks/neuralnetworks.txt

be the set of labels with Q discrete labels. Let N be the number of training examples. Let q be some quality criterion that is used to measure the effectiveness of the learned predictive model. In *multi-label classification* the goal is to find a mapping $h : \mathcal{X} \rightarrow 2^{\mathcal{L}}$ that maximizes q.

Methods used for multi-label classification can be divided in to two main categories [57]: (1) *algorithm adaptation methods* and (2) *problem transformation methods. Algorithm adaptation methods* modify existing machine learning algorithms for multi-label classification. *Problem transformation methods* transform the multi-label classification problem into a one or more single label classification problems. Then binary classifiers can be used to solve these single label classification problems. In this work we used four multi-label learning methods falling in to these two categories. Below we describe these methods.

2.3.2.1. *Pair-wise methods.* Pair-wise methods fall in to the category of problem transformation methods. These methods use binary classifiers in a pair-wise or round robin fashion to solve multi-label classification problems [58]. These methods use $Q(Q-1)/2$ binary classifiers, where each classifier is trained with examples of one label λ_i as positive examples and the examples of another label λ_j as negative examples. During the testing phase, for a given test instance, each classifier predicts one of the two labels. Then the labels are ranked according to their sum of votes. A label ranking algorithm uses these rankings to predict the labels for a given test instance.

In this work we used a method called *calibrated label ranking (CLR)* that extends the pair-wise approach described above [59]. CLR introduces an artificial calibration label λ_0 to represent the split-point between relevant and irrelevant labels i.e. λ_0 is preferred over irrelevant labels and relevant labels are preferred over λ_0 . During testing the classifiers

predict a ranking for the Q+1 labels and labels ranked above λ_0 are considered as relevant labels for the testing instance.

2.3.2.2. *Label power-set methods (LP).* Label power-set methods also fall into the category of problem transformation methods. This method converts the multi-label classification problem into a single-label multi-class classification problem. The set of distinct unique subsets of labels in the training set becomes the possible values of the class. Due to this transformation, LP methods take into account the dependencies among labels. But the number of possible label sets can increase exponentially with this method and can be a problem when used with a large number of labels. Also class imbalance can be a potential problem with this method since some of the newly formed classes might have very few training examples.

In this work we used a label power-set method called a *hierarchy of multi-label classifiers (HOMER)* [60]. HOMER creates a hierarchy of multiple labels and constructs a classifier for the label sets in the nodes of hierarchy. By creating this hierarchy each classifier deals with a more balanced example distribution.

2.3.2.3. *Ensemble methods.* Ensemble methods are developed on top of problem transformation methods or algorithm adaptation methods. In this work we used two ensemble method called *random k-label sets (RAkEL)* [61] and *ensemble classifier chains (ECC)* [62]. RAkEL draws m random label subsets of of size k from $\mathcal{L}$ and trains a LP classifier on each of the k subsets. Final set of labels for a given test instance is selected using a simple voting mechanism. Similar to other LP methods, RAkEL also considers dependencies among labels by using label subsets.

ECC method uses *classifier chains (CC)* [62] as base classifiers. In CC, Q binary classifiers are linked as a chain as $C_1, C_2, ..., C_Q$. The classifier C_i in the chain is responsible for learning

and predicting the binary association of label λ_i where $(1 \leq i \leq Q)$. The Feature space of each classifier C_i in the chain is extended by adding the label values of all previous links in the chain $\lambda_1, ..., \lambda_{i-1}$. The CC method takes into account the label correlations by passing the label information between classifiers as explained above. ECC trains m CCs, where each CC is trained with a random ordering of the labels and a random subset of the data. Therefore each CC is likely to give different multi-label predictions. Then these predictions are summed for each label, and threshold is used to select the most popular labels from the predicted set of labels.

2.3.2.4. *Multi-label k-nearest neighbors (ML-kNN).* ML-kNN [63] is an extension of the k-nearest neighbors (kNN) algorithm [64] for single label classification. For each instance in the test set, ML-kNN first finds the nearest neighbors from the training set. Then using the statistical information calculated from the labels of these nearest neighbors, the label set for the test instance is determined.

2.3.2.5. *BP-MLL.* BP-MLL is neural network algorithm that uses back-propagation for multi-label classification [65]. BP-MLL is derived from the back-propagation algorithm [66] by replacing the error function. The modified error function ranks labels belonging to an instance higher than the labels that do not belong to that instance.

2.3.3. Kernel methods. Kernel methods perform pattern analysis by following two main steps: (1) Embed the data in an appropriate feature space. (2) Use machine learning algorithms to discover linear relations in the embedded data [67]. Figure 2.2 depicts how the data is mapped to a feature space so that a linear separation of the data can be obtained. There are two main advantages of using kernel methods. First, machine learning algorithms for discovering linear relations are efficient and are well understood. Second, a *kernel function*

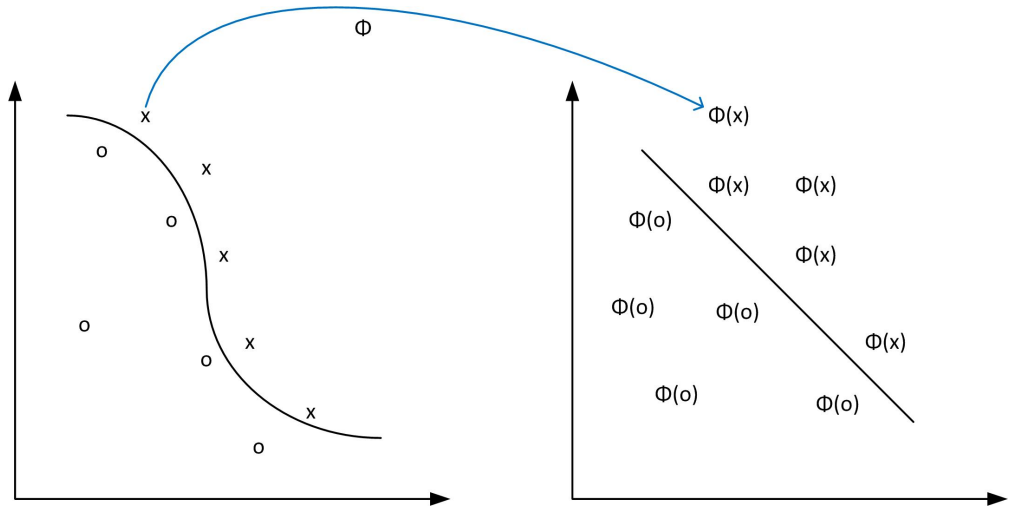


FIGURE 2.2. Function ϕ maps data into a new feature space so that a linear separation can be found.

can be used to compute the inner product of data in the new feature space without explicitly mapping the data into that space. A *kernel function* computes the inner products in the new feature space directly from the original data [67]. Let k be the kernel function and $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m$ be training data. Then the kernel function calculates the inner product in the new feature space using only the original data without having to compute the mapping ϕ explicitly as follows: $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$. Machine learning algorithms such as support vector machines (SVMs) can use the kernel function instead of calculating the actual coordinates in the new feature space.

In this work we represent the program functions using a graph that captures both control flow and data dependency information. Therefore we provide some background on methods used for comparing graphs. Graph comparison/classification has been previously applied in areas such as bioinformatics, chemistry, sociology and telecommunication. Graph comparison algorithms can be divided into three groups [68]: (1) *set based*, (2) *frequent subgraph based* and (3) kernel based. Set based methods compare the similarity between the set of nodes and the set of edges in two graphs. These methods do not consider the topology of the

graph. Frequent subgraph based methods first develop a set of subgraphs that are frequently present in a graph dataset of interest. Then these selected subgraphs are used for graph classification. Frequent subgraph based methods are computationally expensive and the complexity increases exponentially with the graph size. In this dissertation we use the kernel based approach, specifically graph kernels and compare their performance to the set of features we used in our previous work [69].

Graph kernels are a set of kernel functions used for graph data. Graph kernels compute a similarity score for a pair of graphs by comparing their substructures, such as shortest paths [70], random walks [71], and subtrees [72]. Graph kernels provide a method to explore the graph topology by comparing graph substructures in polynomial time. Therefore graph kernels can be used efficiently to compare similarities between programs which are represented as graphs (e.g. control flow graphs and program dependency graphs).

2.3.4. Evaluation measures.

2.3.4.1. *Evaluation measures for binary classification.* We used two evaluation measures to measure the effectiveness of binary classifiers. The first evaluation measure is the *balanced success rate (BSR)* [73]. Standard accuracy gives the fraction of the correctly classified instances in the data set. Therefore it is not a good measure to evaluate success when the data set is unbalanced as in our case. BSR considers the imbalance in data as follows:

$$BSR = \frac{P(success|+)+P(success|-)}{2},$$

where $P(success|+)$ is the estimated probability of classifying a positive instance correctly and $P(success|-)$ is the probability of classifying a negative instance correctly.

The second evaluation measure is the area under the receiver operating characteristic curve (AUC) value. AUC measures the probability that a randomly chosen negative example

will have a smaller estimated probability of belonging to the positive class than a randomly chosen positive example [74]. Therefore a higher AUC value indicates that the model has a higher predictive ability. AUC takes a value in the range [0,1]. A classifier with AUC = 1 is considered a perfect classifier, while a classifier that classifies randomly will have AUC = 0.5. Previous studies have shown that AUC is a better measure for comparing learning algorithms [74]. Since the AUC does not depend on the discrimination threshold of the classifier we use the AUC measure for comparisons.

2.3.4.2. *Evaluation measures for multi-label classification.* We used *label based measures* from Madjarov et al. [56] to evaluate the effectiveness of the multi-label classifiers. We can use these measures to compare the performance of multi-label classifiers with binary classifiers, since these measures can be computed for a set of binary classifiers that predicts the same set of labels. The measures are defined as follows:

- *Macro-precision* is the precision averaged across all the labels and it is defined as

(1) $$macro - precision = \frac{1}{Q}\sum_{j=1}^{Q}\frac{tp_j}{tp_i + fp_j}$$

where tp_j and fp_j are the number of true positives and false positives for the label λ_j when considered as a binary label. Q is the number of unique labels.

- *Macro recall* is the recall averaged across all the labels and it is defined as

(2) $$macro \quad recall = \frac{1}{Q}\sum_{j=1}^{Q}\frac{tp_j}{tp_j + fn_j}$$

where tp_j and fn_j are the number of true positives and false negatives for the label λ_j when considered as a binary label.

- *Macro* F_1 is the harmonic mean between precision and recall and it is defined as

$$macro \quad F1 = \frac{1}{Q}\sum_{j=1}^{Q}\frac{2 \times p_j \times r_j}{p_j + r_j} \tag{3}$$

where p_j and r_j are the precision and recall for the label λ_j.

- *Micro precision* is the precision averaged over all the example/label pairs and it is defined as

$$micro \quad precision = \frac{\sum_{i=1}^{Q} tp_i}{\sum_{i=1}^{Q} tp_i + \sum_{i=1}^{Q} fp_i} \tag{4}$$

- *Micro recall* is the recall averaged over all the example/label pairs and it is defined as

$$micro \quad recall = \frac{\sum_{i=1}^{Q} tp_i}{\sum_{i=1}^{Q} tp_i + \sum_{i=1}^{Q} fn_i} \tag{5}$$

- *Micro* F_1 is the harmonic mean between micro precision and micro recall. It is defined as

$$micro \quad F1 = \frac{2 \times micro \quad precision \times micro \quad recal}{micro \quad precision + micro \quad reca} \tag{6}$$

2.4. Summary

In this chapter, we described the oracle problem in scientific software testing, techniques used to alleviate the oracle problem and their limitations. Then, we provided details of the metamorphic testing technique, previous work done on metamorphic testing and how our work contribute towards state of the art in metamorphic testing. Finally, we provided the

background information for the machine learning techniques that we use in this work. Specifically, we discussed the binary classification algorithms, multi-label classification algorithms and kernel methods used in this work. We also described the evaluation measures used to measure the accuracy of the machine learning prediction models.

CHAPTER 3

Effectiveness of Machine Learning Approaches for Predicting Metamorphic Relations

This Chapter presents an overview of using machine learning techniques for predicting metamorphic relations of a given function [69]. Sections 3.2.1, 3.2.2, and 3.2.3 provide the details of function representation, feature extraction, and prediction steps in our method. We present our evaluation procedure and results in Section 3.3. Finally, we present our conclusions in Section 3.4.

3.1. Motivating example

Figure 3.1 displays a function that finds the maximum value of an array. Figure 3.2 displays a function for setting a minimum value in an array. Figure 3.3 depicts the control flow graphs of the two functions. Consider the permutative metamorphic relation, which states that if the elements in the input are randomly permuted, the output should remain constant. The function *find_max* satisfies the permutative metamorphic relation. But the *set_min_val* function does not satisfy the permutative property since the order of the elements in the output will not be the same after permuting the input.

The overall structure of the two CFGs are very similar. But the main difference is in the operations performed inside the loop. While *find_max* perform a comparison over the array elements, *set_min_val* performs assignments to individual array elements. Therefore it is important to include information about the sequence of operations performed in a control flow path in the features used for creating the machine learning prediction models. Based on

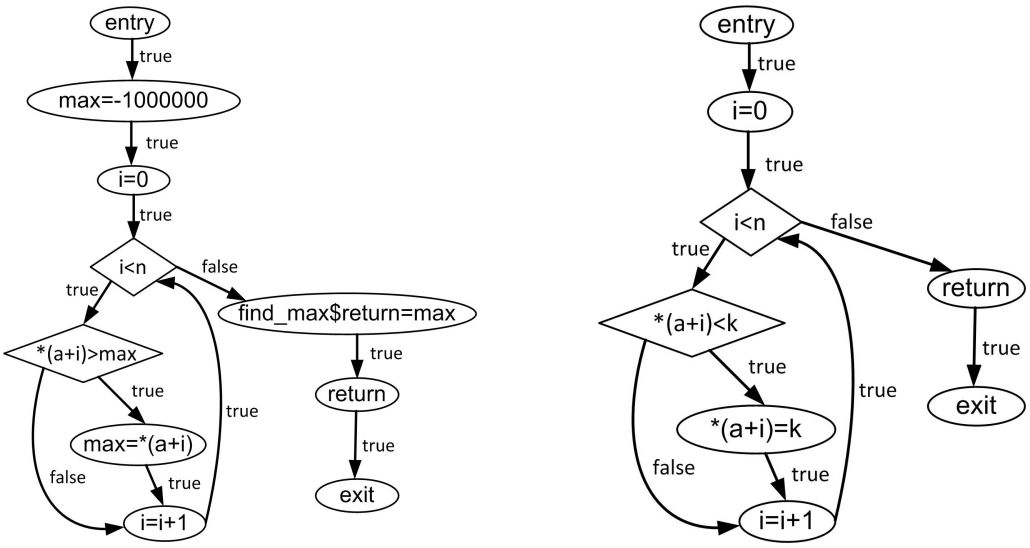
this intuition, we developed two types of features for CFGs, which are based on the nodes and paths in the CFG. We describe these features in Sections 3.2.2.

```
int find_max(int a[],int n){
        int max=-1000000,i;
        for(i=0;i<n;i++){
                if(a[i]>max){
                        max=a[i];
                }
        }
        return max;
}
```

FIGURE 3.1. find_max

```
void set_min_val(int a[],int n,int k){
        int i;
        for(i=0;i<n;i++){
                if(a[i]<k){
                        a[i]=k;
                }
        }
}
```

FIGURE 3.2. set_min_val



(A) find_max (B) set_min_val

FIGURE 3.3. CFGs of find_max and set_min_val

Figure 3.4 shows an overview of our method. We start by creating the *control flow graph (CFG)* from a function's source code. Next, we extract a set of features from the CFGs, and a machine learning algorithm uses these features to create a predictive model. Finally, we use the developed predictive model to predict the metamorphic relations in previously unseen functions.

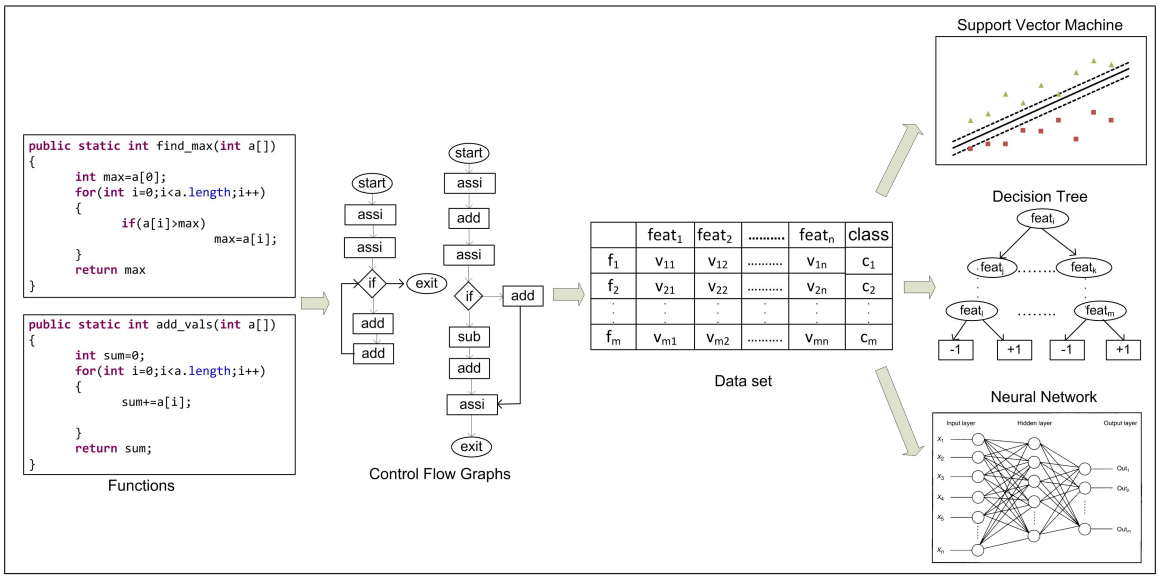
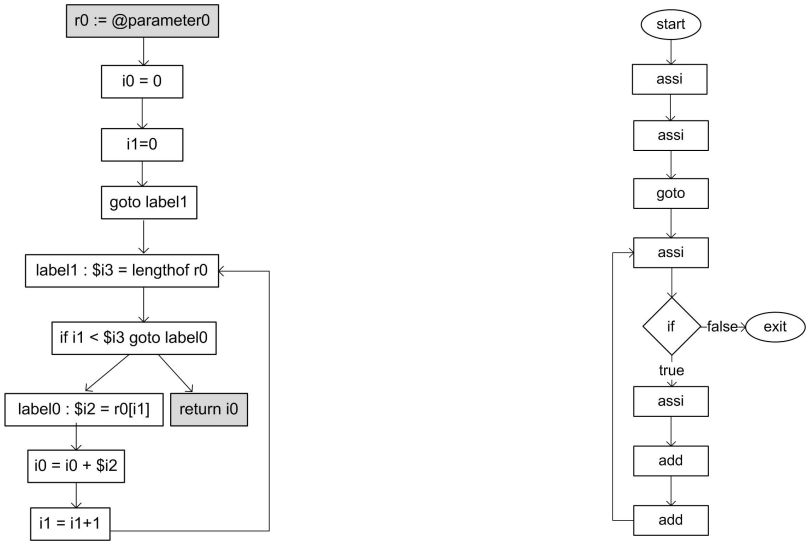


FIGURE 3.4. Overview of the proposed method

3.2. METHOD OVERVIEW

3.2.1. FUNCTION REPRESENTATION. We hypothesize that the metamorphic relations in Table 3.3 are related to the sequence of operations performed by a function. Therefore, we represent a function using a statement level CFG, since it models the sequence of operations. The CFG $G = (N, E)$ of a function f is a directed graph, where each $n_x \in N$ represents a statement x in f and $E \subseteq N \times N$. An edge $e = (n_x, n_y) \in E$ if x, y are statements in f and y is executed immediately after executing x. Nodes $n_{start} \in N$ and $n_{exit} \in N$ represents the starting and exiting points of f.

We used the *Soot*[1] framework to create CFGs. *Soot* generates control flow graphs in *Jimple* [75], a typed 3-address intermediate representation, where each CFG node represents an atomic operation. This representation should considerably reduce the effects of different programming styles and models the actual control flow structure of the function. Figure 3.5a is the *Jimple* statement level CFG generated using the *Soot* framework for the function in Figure 2.1. Converting Java code to the Jimple 3-address intermediate representation would add goto operations and labels to represent conditional jumps in the original Java code. Then a *labeled CFG* is created by giving a label to each node in the CFG in Figure 3.5a to indicate the operation performed in the node. Figure 3.5b is the labeled CFG created from the original CFG in Figure 3.5a.



(A) *Soot* CFG (B) Labeled CFG

FIGURE 3.5. CFG generated by *Soot* with 3-address code and Labeled CFG for the program in Figure 2.1

3.2.2. FEATURE EXTRACTION. We extracted two types of features based on the nodes and paths in the CFG.

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

3.2.2.1. *Node Features.* For a CFG, node features have the form $op - d_{in} - d_{out}$, where *op* is the operation performed in a node $n \in N$, d_{in} is the *in-degree* of n and d_{out} is the *out-degree* of n. The value for a given feature is the number of occurrences of nodes of type $op - d_{in} - d_{out}$ in the CFG. Table 3.1 shows the node features calculated for the labeled graph in Figure 3.5b.

TABLE 3.1. Node features calculated from the labeled graph in Figure 3.5B

Feature	**Feature Value**
start-0-1	1
if-1-2	1
add-1-1	2
assi-1-1	3
assi-2-1	1
goto-1-1	1
exit-1-0	1

3.2.2.2. *Path Features.* Features based on paths are created by taking the sequence of nodes in the shortest path from N_{start} to each node and the sequence of nodes in the shortest path from each node to N_{exit}. A path feature takes the form $op_1 - op_2 - ... - op_k$ where $op_i (1 \leq i \leq k)$ represents the operation performed in the CFG nodes in the considered path. The value of a path feature is the number of occurrences of that shortest path node sequence in the CFG. Table 3.2 shows the set of features extracted from the labeled graph in Figure 3.5b. For the example considered here, each node sequence occurs only once in the labeled graph in Figure 3.5b. Therefore each feature value for this example takes the value one.

3.2.3. PREDICTION. In this work we focus on predicting whether a given function f exhibits a metamorphic relation in Table 3.3. We selected the *permutative*, *additive* and *inclusive* metamorphic relations for this experiment. These relations represent three different

TABLE 3.2. Path features calculated from the labeled graph in Figure 3.5B

Feature	**Feature Value**
start-assi-assi-goto-assi-if-assi-add	1
start-assi-assi-goto-assi-if-assi	1
start-assi-assi	1
start	1
start-assi-assi-goto-assi-if	1
start-assi-assi-goto-assi-if-exit	1
start-assi-assi-goto-assi	1
start-assi	1
start-assi-assi-goto-assi-if-assi-add-add	1
start-assi-assi-goto	1
assi-goto-assi-if-exit	1
exit	1
goto-assi-if-exit	1
assi-if-exit	1
if-exit	1
assi-add-add-assi-if-exit	1
add-assi-if-exit	1
assi-assi-goto-assi-if-exit	1
add-add-assi-if-exit	1

categories of input modifications: (1) changing the order of elements (permutative), (2) changing the element values (additive, multiplicative and invertive) and (3) adding/removing new element/s to/from the input (inclusive and exclusive). The three selected metamorphic relations represent a diverse initial set of relations for use in evaluating our method. Although there could be a variety of changes in the output when the input is modified using these relations, we have considered only the specific output changes given in Table 3.3 for each metamorphic relation.

We modeled this problem as a machine learning classification problem, where each function f has a class label with the value 1 or -1 depending on whether f exhibits a specific metamorphic relation or not, respectively. Table 3.4 depicts an example data set used for learning the classifier, where f_i represents a function in the data set and $feat_j$ represents

TABLE 3.3. The metamorphic relations used in this study.

Relation	**Change made to the input**	**Expected change in the output**
Permutative	Randomly permute the elements	Remain constant
Additive	Add a positive constant	Increase or remain constant
Multiplicative	Multiply by a positive constant	Increase or remain constant
Invertive	Take the inverse of each element	Decrease or remain constant
Inclusive	Add a new element	Increase or remain constant
Exclusive	Remove an element	Decrease or remain constant

TABLE 3.4. Example Data set used for prediction

Function	**feat$_1$**	**feat$_2$**	...	**feat$_n$**	**Class**
f_1	v_{11}	v_{22}	...	v_{1n}	c_1
f_2	v_{11}	v_{22}	...	v_{2n}	c_2
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
f_m	v_{m1}	v_{m2}	...	v_{mn}	c_m

a node or path feature extracted from the labeled CFGs of functions. The feature value of $feat_j$ for the function f_i is represented by v_{ij} ; c_i represents the class label for the function f_i indicating whether f_i exhibits a specific metamorphic relation or not. Three data sets were created for each metamorphic relation in used in the experiments and they were used as input to the SVM, decision tree, and ANN classification algorithms.

3.3. EVALUATION

3.3.1. DATA SET. To measure the effectiveness of our proposed method, we built a code corpus containing 48 mathematical functions that take numerical inputs and produce numerical outputs. None of these functions have an oracle to check the correctness of the output for a randomly generated input. Table A.1 and Table A.2 (in Appendix A) shows the details of the functions used in the experiment. These functions were implemented using the Java programming language.

3.3.2. Evaluation Procedure. Our evaluation procedure is two fold. We measured the effectiveness of (1) our predictive model and (2) the predicted metamorphic relations in detecting faults. The latter was conducted to validate the usefulness of the metamorphic relations predicted by our method.

3.3.2.1. *Predictive Model Evaluation.* We used the two evaluation measures: BSR and AUC described in Section 2.3.4. We used *stratified 10-fold cross-validation* to evaluate the performance of classification. The *10-fold cross-validation* technique evaluates how a predictive model would perform on previously unseen data. In *10-fold cross-validation* the data set is randomly partitioned into ten subsets. Then nine subsets are used to build the predictive model (training) and the remaining subset is used to evaluate the performance of the predictive model (testing). This process is repeated k times in which each of the k subsets is used to evaluate the performance. In *stratified 10-fold cross-validation*, *10* folds are partitioned in such a way that the folds contain approximately the same proportion of positive (functions that exhibit a specific metamorphic relation) and negative (functions that do not exhibit a specific metamorphic relation) examples as in the original data set.

3.3.2.2. *Fault Detection Effectiveness.* We used mutation analysis [76] to measure the effectiveness of the predicted metamorphic relations from our method in detecting faults. Mutation analysis operates by inserting faults into the program under test such that the created faulty version is very similar to the original version of the program [77]. A faulty version of the program under test is called a *mutant*. If a test identifies a mutant as faulty that mutant is said to be *killed*.

Mutation analysis was conducted on 35 functions from Table A.3, which exhibits all or several of the three relations in Table 3.3. We used the μJava[2] mutation engine to create

[2]http://cs.gmu.edu/∼offutt/mujava/

TABLE 3.5. Summary of the data sets used for prediction

Metamorphic Relation	**#Positive**	**#Negative**	**Total**
Permutative	20	21	41
Additive	21	15	36
Inclusive	15	12	27

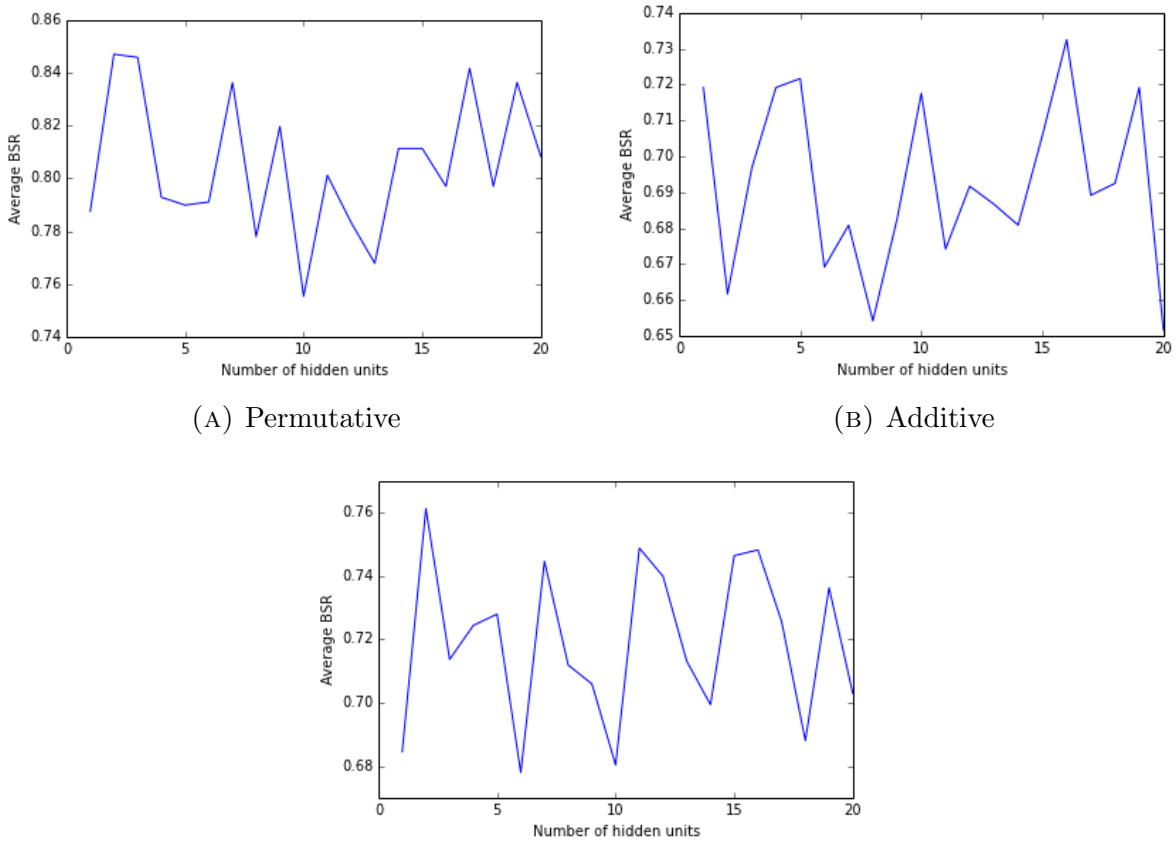
the mutants for the functions in our code corpus. We used only method level mutation operators [78] to create mutants since we are only interested in the faults at the function level. Each mutated version of a function was created by inserting only a single mutation. Mutants that resulted in compilation errors, run-time exceptions or infinite loops were removed before conducting the experiment.

For each function f we randomly generated 10 initial test cases. We then created follow-up test cases using the metamorphic relations shown by f, for each of the initial test cases. Finally we checked whether the corresponding metamorphic relations were satisfied by the initial and follow-up test case pairs. A mutant of f is killed, if at least one pair of test cases fail to satisfy the corresponding metamorphic relation m.

3.3.3. PREDICTIVE MODEL EVALUATION RESULTS. We conducted the evaluation of the prediction of the metamorphic relations in Table 3.3. Table 3.5 gives the number of positive and negative examples contained in each of the data sets used for the evaluation.

3.3.3.1. *Performance of ANNs.* When using ANNs it is required to test with different ANN structures. We present the results of this evaluation first. First, we evaluated the effectiveness of ANNs with a single hidden layer for metamorphic relation prediction. Figure 3.6 shows the variation of average BSR with the number of hidden units in the hidden layer. The number of hidden units that gave the maximum BSR varied with the MR. Table 3.6 summarizes maximum average BSR, the standard deviation of the average BSR and the

number of hidden units that gave the maximum average BSR for each MR. For permutative and inclusive MRs the maximum BSR could be archived using only two hidden units. For the other metamorphic relation the number of hidden units required were 16.



(A) Permutative

(B) Additive

(C) Inclusive

FIGURE 3.6. Variation of BSR with the number of hidden units in NNs with a single layer

TABLE 3.6. Performance of single layered NNs

MR	Maximum BSR	SD	#Hidden units
Permutative	0.847	0.101	2
Additive	0.732	0.135	16
Inclusive	0.761	0.128	2

Secondly, we evaluated the performance of NNs with multiple hidden layers. We used the hidden layer structures shown in the following list in our experiment. Each item in the list

corresponds to a single hidden layer structure. The length of the number sequence gives the number of hidden layers. Each number in the number sequence gives the number of hidden units in the corresponding hidden layer. Table 3.7 displays the structure of the multi-layered NN that gave the maximum BSR for each MR.

- (2)
- (2,2)
- (2,2,2)
- (2,2,2,2)
- (2,2,2,2,2)
- (3)
- (3,3)
- (3,3,3)
- (3,3,3,3)
- (3,3,3,3,3)
- (4)
- (4,4)
- (4,4,4)
- (4,4,4,4)
- (4,4,4,4,4)
- (2,3,4,5,6)

TABLE 3.7. Performance of multi-layered NNs

MR	**Maximum BSR**	**SD**	**Multi-layered NN structure**
Permutative	0.879	0.08	(3,3)
Additive	0.728	0.151	(4,4,4,4,4)
Inclusive	0.743	0.128	(3,3,3)

3.3.3.2. *Performance comparison of ML algorithms.* We report the performance comparison of the three ML algorithms in this section. We report only the BSR values for the ANNs since the ANN implementation we used in this work did not provide facilities to compute AUC values. For the SVM and decision tree models we report both the BSR and the AUC values. Table 3.8 reports these results. Reported values in the Table 3.8 are the average BSR and AUC values for the 10 cross validation runs with the standard deviation reported inside parenthesis. Inital evaluation showed that linear kernel performas better than the polynomal and gaussian kernels. Therfore here we report only the results for the SVM obtained using the linear kernel (default parameters (C=10) provided by PyML. For ANNs we used one hidden layer with two hidden units for this evalutation.

As shown in Figure 3.7 and 3.8, when compared with the decision tree model and ANNs, SVM gives an overall better performance with respect to BSR and AUC measures for all three metamorphic relations. This is expected because SVMs tend to perform better than other machine learning methods since they are less prone to over-fitting [79]. For all the three relations used for prediction, SVM achieves an accuracy higher than 0.8 and AUC higher than 0.9. The high accuracy and AUC achieved by the SVM model for the three metamorphic relations shows that the feature set developed using the CFGs can effectively predict metamorphic relations.

TABLE 3.8. Prediction results from SVMs, Decision Trees, and ANNs

Metamorphic Relation	**BSR**			**AUC**	
	SVM	DT	NN	SVM	DT
Permutative	0.89 (0.04)	0.87 (0.03)	0.85 (0.10)	0.93 (0.03)	0.81 (0.03)
Additive	0.83 (0.04)	0.78 (0.05)	0.66 (0.13)	0.90 (0.04)	0.82 (0.04)
Inclusive	0.87 (0.02)	0.73 (0.02)	0.76 (0.13)	0.94 (0.03)	0.66 (0.03)

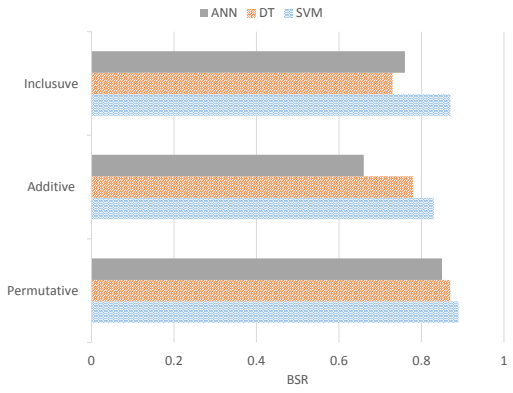


FIGURE 3.7. BSR for SVM, Decision Tree, and ANN models for predicting Permutative, Additive and Inclusive Metamorphic Relations

Secondly, we evaluated how the performance of a classifier varies with the training set size when predicting metamorphic relations. Using a large number of examples to train a classifier will reduce the chance of the classifier being biased [80]. But using a large training

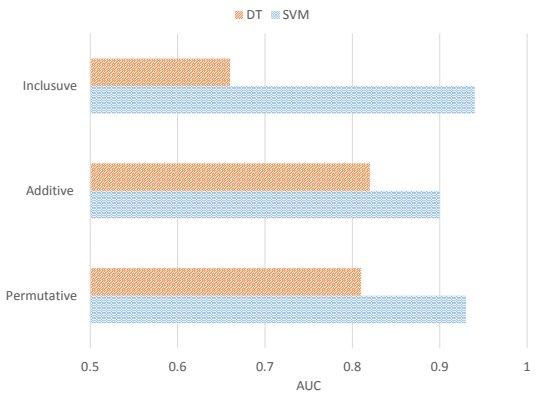


FIGURE 3.8. AUC for SVM and Decision Tree models for predicting Permutative, Additive and Inclusive Metamorphic Relations

set will increase the cost of this approach since it requires identifying metamorphic relations for the programs in the training set manually. Therefore we investigated whether effective classifiers can be learned using a small set of programs.

We used SVMs in this evaluation since they performed significantly better than decision trees and neural networks. We first randomly partitioned each dataset in Table 3.5 into two sets so that each half would contain approximately the same proportions of positive and negative examples as the original dataset. One partition was used as the test set. From the other partition we randomly selected training examples to train the classifier. For each training set size we tested the trained classifier using all instances in the test set.

Figures 3.9, 3.10 and 3.11 show the variation of the accuracy and AUC of the classifiers for different training set sizes in predicting permutative, additive and inclusive metamorphic relations respectively. As expected, the accuracy and AUC of the classifiers increase with the training set size. For the three metamorphic relations, even classifiers built using the smallest possible training set perform better than a random classifier. For the permutative relation, the classifier achieves a 0.8 AUC when trained with a set of five programs. For the inclusive relation, even the smallest training set containing only three programs achieved

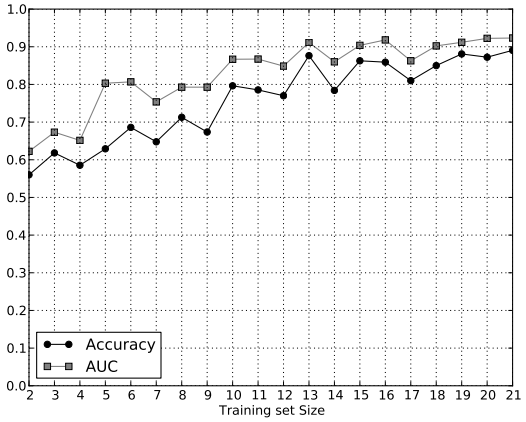


FIGURE 3.9. Variation of accuracy and AUC with training set size for predicting permutative MR

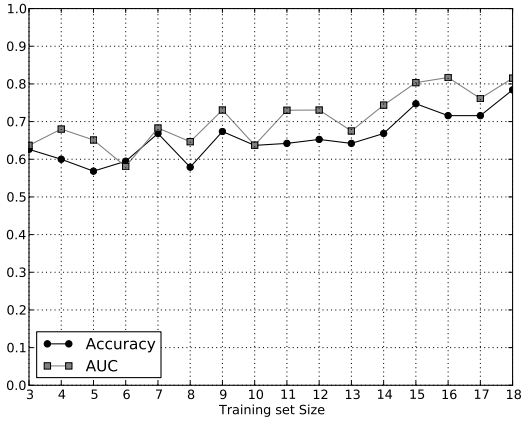


FIGURE 3.10. Variation of accuracy and AUC with training set size for predicting additive MR

an AUC of 0.8. For the additive relation, a 0.8 AUC was achieved with a training set of 15 programs. These result show that our method works effectively with a small number of training examples.

Finally we evaluated the performance of the classifiers when the prediction is done on programs that contain at least one fault. In practice, a test engineer may have a set of programs that satisfies known metamorphic relations. The engineer can apply our prediction method to determine which of these metamorphic relations should be satisfied by a new,

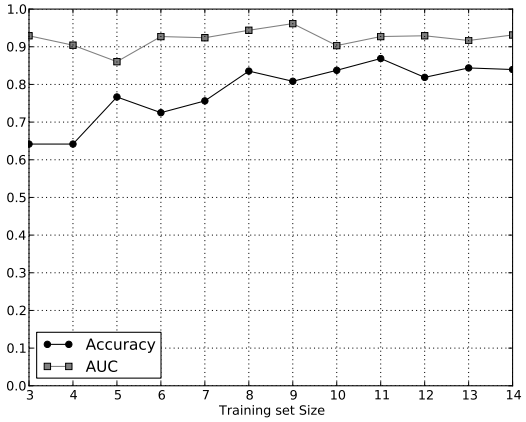


FIGURE 3.11. Variation of accuracy and AUC with training set size for predicting inclusive MR

possibly faulty program. To evaluate how our method works when applied to such a scenario, we did the following for each function f in the data set:

(1) We removed f from the data set and trained the SVM classifier using the remaining functions.

(2) We generated predictions for the original function f and for randomly selected mutants of function f using the classifier produced in Step 1.

(3) We compared the classification for each mutant to the classification for the original function.

Table 3.9 shows the results of this evaluation. The third column shows the number of mutants that were classified differently than the original function. The classification of the mutants did not match that of the original function for only 1%-5% of the mutants, depending on the metamorphic relation. These results indicate that the prediction model can provide a reasonably accurate classification for functions that contain a fault.

3.3.4. FAULT DETECTION EFFECTIVENESS RESULTS. Table 3.10 gives a summary of the mutants generated for the mutation analysis performed to evaluate the effectiveness

TABLE 3.9. Results of predicting metamorphic relations for the faulty versions of the functions

Metamorphic Relation	**# Mutants used**	**# Classified differently**
Permutative	171	3
Additive	146	7
Inclusive	131	2

TABLE 3.10. Summary of mutants used in the mutation analysis

# Mutants generated by *mu*Java	1717
# Mutants resulted in Exceptions	591
# Mutants resulted in Infinite loops	138
# Mutants used in the experiment	988

of our approach in detecting faults. After removing mutants that are obviously incorrect (mutants that gave exceptions and infinite loops), 988 mutants in total were used for the experiment.

Out of 988 mutants, 655 (66%) were killed using the predicted metamorphic relations. More than 50% of the mutants were killed in 29 functions. This shows that we can apply these predicted relations successfully to detect faults. Table 3.11 shows the percentages of mutants that were killed using each metamorphic relation alone. The permutative relation has the highest percentage of killed mutants. This result was expected since, for the functions studied in this experiment, the permutative metamorphic relation requires the outputs of the initial and the follow-up test cases to be equal (see Table 3.3). This equality relation is more restrictive and thus can be more easily violated than the inequality relation of the additive and inclusive metamorphic relations.

TABLE 3.11. Percentage of mutants killed by each metamorphic relation

Metamorphic Relation	**# Mutants possible to kill**	**# Mutants killed (%)**
Permutative	566	374 (66%)
Additive	869	196 (23%)
Inclusive	400	150 (38%)

Several mutants could not be killed using any of the predicted metamorphic relations. Some of these mutants are making changes that could not be captured by the metamorphic relations that we used in this study. Additional metamorphic relations might be needed to kill them. Some of the survivors are equivalent mutants that cannot be killed since they produce the same output as the original program [77].

3.4. Conclusions

We have presented a novel machine learning approach to automatically detect likely metamorphic relations of program functions using features extracted from a function's control flow graph. We have evaluated the performance of our predictive model using a set of real world functions that do not have oracles. Overall, SVMs performed significantly better than decision trees and ANNs. High accuracy and AUC achieved by the SVM predictive model show that features developed using the CFGs are highly effective in predicting metamorphic relations. We also showed that our method can create effective classifiers using reasonably small training sets making this approach cost effective for use in practice. Further, we show that, when applied to programs with an injected fault, our method produces the same predictions as those produced for the original program in at least 95% of the cases. Thus, the identified metamorphic relations should be accurate even for faulty programs. Finally, using mutation analysis we showed that the predicted relations can effectively detect faults.

CHAPTER 4

Effectiveness of Graph Kernels for Predicting Metamorphic Relations

In this Chapter we extended the approach introduced in Chapter 3 using graph kernels for feature extraction. In Section 4.1 we provide an overview on how the graph kernels fit into our approach. In Chapter 3 we showed that node/path features computed using the control flow graphs are effective in predicting metamorphic relations. The first graph kernel that we use is called the *random walk kernel* and it uses the paths in a graph for computing the kernel value. Compared to node/path features, the random walk kernel allows you to incorporate additional information about the types of operations represented by nodes in the graph. We present the details of the random walk kernel in Section 4.2.1. Then we evaluated the effectiveness of a different graph substructure: subgraphs for predicting metamorphic relations. For extracting subgraph features we used the *graphlet kernel*. In addition, we also compared the effectiveness of features that would represent the control flow and data dependency information of a function for predicting metamorphic relations. We describe the experimental setup and the results in Section 4.3 and Section 4.4, respectively.

4.1. Method overview

Figure 4.1 shows an overview of our approach. During the *training phase*, we start by creating a graph based representation that shows both the control flow and data dependency information of the functions in the *training set*, which is a set of functions associated with a label that indicates if a function satisfies a given metamorphic relation (positive example) or not (negative example). Then we compute the graph kernel values that give a similarity

score for each pair of functions in the training set. Then the computed graph kernel is used by an SVM to create a predictive model. During the *testing phase*, we use the developed model to predict whether a previously unseen function satisfies the considered metamorphic relation.

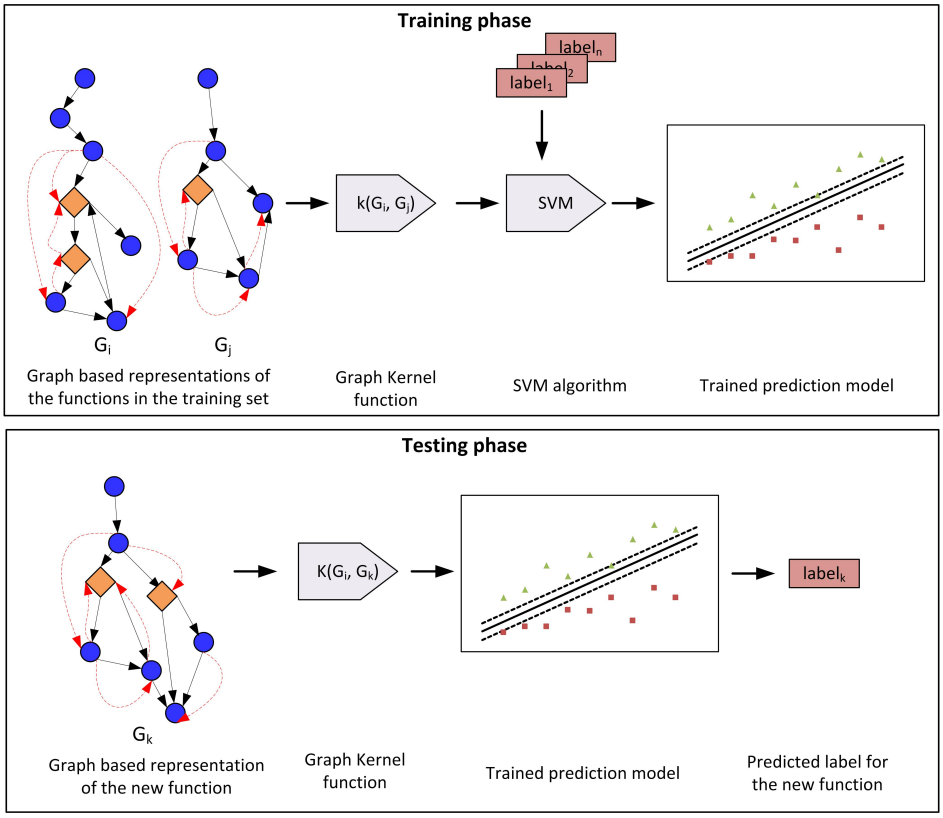


FIGURE 4.1. Overview of the approach. During the training phase a set of functions associated with a label representing the satisfiability of an MR is used for training an SVM classifier that uses graph kernel values computed using the graph representations of these functions. During the testing phase the trained model is used to predict whether a previously unseen function satisfies the MR.

4.1.1. FUNCTION REPRESENTATION. We extended the graph based representation used in the previous Chapter to include both control flow information and data dependency information: $G_f = (V, E)$ of a function f is a directed graph, where each $v_x \in V$ represents a statement x in f. Each node is labeled with the operation performed in x, denoted by

$label(v_x)$. An edge $e = (v_x, v_y) \in E$ if x, y are statements in f and y can be executed immediately after executing x. These edges represent the control flow of the function. An edge $e = (v_x, v_y) \in E$ if x, y are statements in f and y uses a value produced by x. These edges represent the data dependencies in the function. The label of an edge (v_x, v_y) is denoted by $label(v_x, v_y)$ and it can take two values: "cfg" or "dd" depending on whether it represents a control flow edge or a data dependency edge, respectively. Nodes $v_{start} \in V$ and $v_{exit} \in V$ represent the starting and exiting points of f [81].

We created this graph-based representation by first creating the CFG using the *Soot*[1] framework. Then we compute the definitions and the uses of the variables in the function and use that information to augment the CFG with edges representing data dependencies in the function. Figure 4.2 displays the graph based representation created for the function in Figure 2.1.

4.2. Graph Kernels

We define two graph kernels for the graph representations of the functions presented in Section 4.1.1: the *random walk kernel* and the *graphlet kernel*. Each kernel captures different graph substructures as described next.

4.2.1. Random walk kernel. Random walk graph kernels [82, 71] count the number of matching walks in two labeled graphs. In what follows we explain the idea, while the details are provided in Appendix B.1. The value of the random walk kernel between two graphs is computed by summing up the contributions of all walks in the two graphs. A walk in a graph $G = (V, E)$ is a sequence of nodes $(v^1, v^2, ..., v^{n-1}, v^n)$ such that $v^i \in V$ and $(v^i, v^{i+1}) \in E$. A *step* in a walk is two consecutive nodes in the walk. Each pair of walks is

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

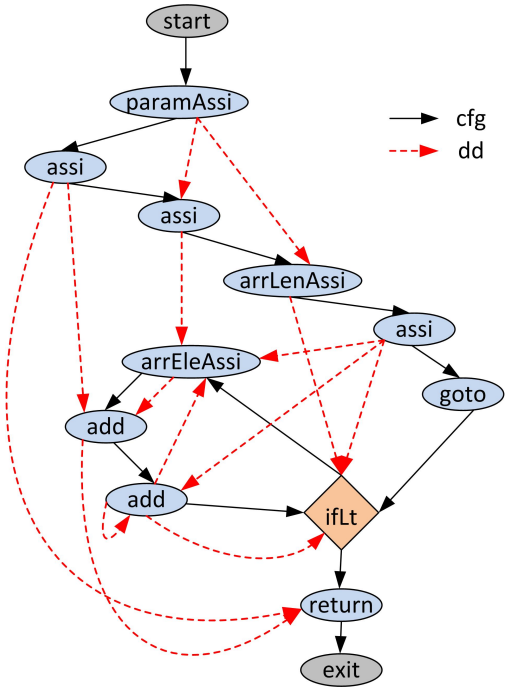


FIGURE 4.2. Graph representation of the function in Figure 2.1. cfg: control flow edges. dd: data dependency edges.

compared using a kernel that computes the similarity of each step in the two walks, where the similarity of each step is a product of the similarity of its nodes and edges. This concept is illustrated in Figure 4.3. Computing this kernel requires specifying an edge kernel and a node kernel. We used two approaches for determining the kernel value between a pair of nodes. In the first approach, we assign a value of one to the node kernel value if the two node labels are identical, and zero otherwise. In the second approach, we assign a value of 0.5, if the node labels represent two operations with similar properties, even if they are not identical (Section B.1 equation (11)). The kernel value between pair of edges is determined using their edge labels, where we assign a value of one if the edge labels are identical zero otherwise.

4.2.2. GRAPHLET KERNEL. Random walks represent sequences of nodes of varying length and do not capture subgraphs in a graph. Subgraphs can directly capture the structure of *if*

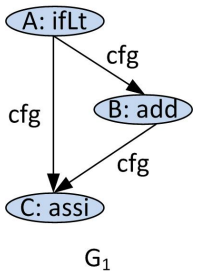
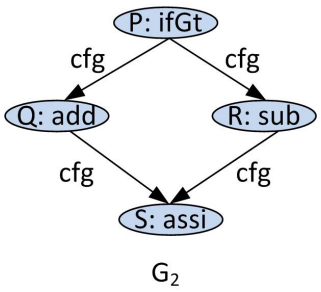
	
Walks of length 1: $A \to B$, $B \to C$, $A \to C$	Walks of length 1: $P \to Q$, $P \to R$, $Q \to S$, $R \to S$
Walks of length 2: $A \to B \to C$	Walks of length 2: $P \to Q \to S$, $P \to R \to S$
Computation of similarity score between two graphs (restricted to walks up to length 2): $k_{rw}(G_1, G_2) = k_{walk}(A \to B, P \to Q) + k_{walk}(A \to B, P \to R) + ... + k_{walk}(A \to B \to C, P \to Q \to S) + k_{walk}(A \to B \to C, P \to R \to S)$	
Computation of similarity score between two walks: $k_{walk}(A \to B, P \to Q) = k_{step}((A, B), (P, Q))$ ... $k_{walk}(A \to B \to C, P \to R \to S) = k_{step}((A, B), (P, R)) \times k_{step}((B, C), (R, S))$	
Computation of similarity score between two steps: $k_{step}((A, B), (P, Q)) = k_{node}(A, P) \times k_{node}(B, Q) \times k_{edge}((A, B), (P, Q))$ $k_{step}((A, B), (P, R)) = k_{node}(A, P) \times k_{node}(B, R) \times k_{edge}((A, B), (P, R))$ $k_{step}((B, C), (R, S)) = k_{node}(B, R) \times k_{node}(C, S) \times k_{edge}((A, B), (P, Q))$	
Computation of similarity score between two nodes: $k_{node}(A, P) = 0.5$ (two labels have similar properties) $k_{node}(B, Q) = 1$ (two labels are identical) $k_{node}(B, R) = 0$ (two labels are dissimilar) $k_{node}(C, S) = 1$	
Computation of similarity score between two edges: $k_{edge}((A, B), (P, Q)) = 1$ (two edges have the same labels) $k_{edge}((A, B), (P, R)) = 1$	

FIGURE 4.3. Random walk kernel computation for the graphs G_1 and G_2. k_{rw}: kernel value between two graphs. k_{walk}: kernel value between two walks. k_{step}: kernel value between two steps. k_{node}: kernel value between two nodes. k_{edge}: kernel value between two edges.

conditions in a function that represent important semantic information about the function. Therefore we apply a kernel based on subgraphs.

The graphlet kernel computes a similarity score of a pair of graphs by comparing all subgraphs of limited size in the two graphs [68]. In this work we use connected subgraphs

with size $k \in \{3, 4, 5\}$ nodes. These subgraphs are called *graphlets*. Consider the pair of graphs G_3 and G_4 in Figure 4.4. The graphlet kernel value of a pair of graphs is calculated by summing up the kernel values of all the graphlet pairs in the two graphs. The kernel value of a pair graphlets is calculated as follows: for each pair of graphlets that are isomorphic, we compute the kernel value by multiplying the kernel values between the node and edge pairs that are mapped by the isomorphism function. If a pair of graphlets are not isomorphic, we assign a value of zero to the kernel value of those two graphlets. The kernel value between pairs of nodes and pairs of edges are determined as explained in Section 4.2.1. In Figure 4.4 we illustrate the computation of the kernel and the complete definition of the graphlet kernel is presented in Appendix B.2.

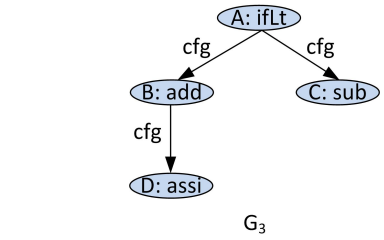
4.3. Experimental setup

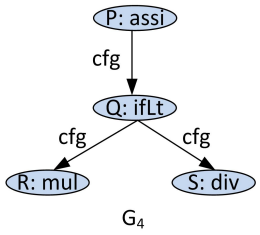
This section describes our research questions and how they will be answered empirically.

4.3.1. Research questions.

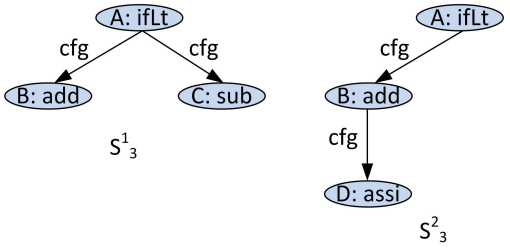
We conducted experiments to seek answers for the following research questions:

- **RQ1: Are graph kernels more effective in predicting metamorphic relations than node/path features?** In Chapter 3 we used a set of features that represent node/path information of control flow graphs for creating predictive models [69]. In this Chapter we extend the feature extraction using graph kernels. We created separate models using the node/path features used in our previous study and the two graph kernels and compared the effectiveness.

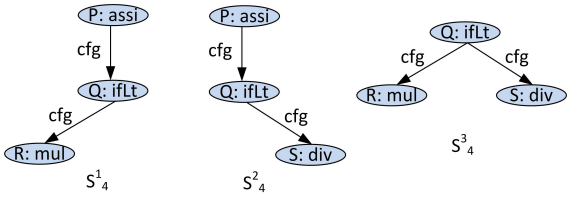




Graphlets of size 3 in G_3:



Graphlets of size 3 in G_4:



Computation of similarity score between two graphs using graphlets:
$k_{graphlet}(G_1, G_2) = k_{subgraph}(S_3^1, S_4^1) + k_{subgraph}(S_3^1, S_4^2) + k_{subgraph}(S_3^1, S_4^3) + ... + k_{subgraph}(S_3^2, S_4^2) + k_{subgraph}(S_3^2, S_4^3)$

Computation of similarity score between two graphlets:
$k_{subgraph}(S_3^1, S_4^1) = 0$
$k_{subgraph}(S_3^1, S_4^2) = 0$
$k_{subgraph}(S_3^1, S_4^3) = k_{node}(A, Q) \times k_{node}(B, R) \times k_{node}(C, S) \times k_{edge}((A, B), (Q, R)) \times k_{edge}((A, C), (Q, S))$
...
$k_{subgraph}(S_3^2, S_4^2) = k_{node}(A, P) \times k_{node}(B, Q) \times k_{node}(D, S) \times k_{edge}((A, B), (P, Q)) \times k_{edge}((B, D), (Q, S))$
$k_{subgraph}(S_3^2, S_4^3) = 0$

Computation of similarity score between two nodes:
$k_{node}(A, Q) = 1$
$k_{node}(B, R) = 0.5$
$k_{node}(C, S) = 0.5$
...
$k_{node}(D, S) = 0$

Computation of similarity score between two edges:
$k_{edge}((A, B), (Q, R)) = 1$
...
$k_{edge}((B, D), (Q, S)) = 1$

FIGURE 4.4. Graphlet kernel computation for graphs G_3 and G_3. $k_{graphlet}$: graphlet kernel value. $k_{subgraph}$: kernel value of between two subgraphs. k_{node}: kernel value between two nodes. k_{edge}: kernel value between two steps.

- **RQ2: What substructures in graphs are more suitable for predicting metamorphic relations?** This is answered by using two graph kernels that use small subgraphs or all walks in the graphs that represent the functions.
- **RQ3: Is control flow information more effective in predicting metamorphic relations than data dependency information?** The two previous research questions focused on feature extraction. Prediction accuracy can also depend on the source of program information used to create the model. In this work we use two types of information about the program: control flow information and data dependency information. We compare the effectiveness of these two types of information for predicting metamorphic relations.
- **RQ4: Will combining control flow information with data dependency information improve the prediction effectiveness of metamorphic relations?** We further look at whether creating prediction models that use both control flow and data dependency information can improve prediction effectiveness.

4.3.2. The code corpus. To measure the effectiveness of our proposed methods, we built a code corpus containing 100 functions that take numerical inputs and produce numerical outputs. We extended the corpus used in the previous Chapter with functions from the following open source projects:

(1) The Colt Project[2]: set of open source libraries written for high performance scientific and technical computing in Java

(2) The Apache Mahout[3]: a machine learning library written in Java

[2]http://acs.lbl.gov/software/colt/
[3]https://mahout.apache.org/

(3) The Apache Commons Mathematics Library[4]: library of mathematics and statistics components written in the Java

We list these functions in Table A.3 in Appendix A[5]. Table 4.1 shows the statistics about the LOC and cyclomatic complexity [83] for the functions in the code corpus.

TABLE 4.1. Size and cyclomatic complexity of the functions in the code corpus

	LOC	**Cyclomatic complexity**
Average	12.37	3.01
Standard deviation	6.6	1.4
Median	10	3
Minimum	5	1
Maximum	54	11

4.3.3. METAMORPHIC RELATIONS. We used the six metamorphic relations shown in Table 4.2. These metamorphic relations were identified by Murphy et al. [84] and are commonly found in mathematical functions. We list the input modifications and the expected output modification of these metamorphic relations in Table 4.2. A function f is said to satisfy (or exhibit) a metamorphic relation m in Table 4.2, if the change in the output is according to what is expected after modifying the original input. Previous studies have shown that these are the type of metamorphic relations that tend to be identified by humans [85–87]. In this work we use the term *positive instance* to refer to a function that satisfies a given metamorphic relation and the term *negative instance* to refer to a function that does not satisfy a considered metamorphic relation. Table 4.3 reports the number of positive and negative instances for each metamorphic relation.

4.3.4. EVALUATION PROCEDURE. We used 10-fold stratified cross validation in our experiments. We used nested cross validation to select the regularization parameter (C) of the

[4]http://commons.apache.org/proper/commons-math/

[5]These functions and their graph representations can be accessed via the following URL: `http://www.cs.colostate.edu/~upuleegk/data/functions.tar.gz`

TABLE 4.2. The metamorphic relations used in this study.

Relation	**Change made to the input**	**Expected change in the output**
Permutative	Randomly permute the elements	Remain constant
Additive	Add a positive constant	Increase or remain constant
Multiplicative	Multiply by a positive constant	Increase or remain constant
Invertive	Take the inverse of each element	Decrease or remain constant
Inclusive	Add a new element	Increase or remain constant
Exclusive	Remove an element	Decrease or remain constant

TABLE 4.3. Number of positive and negative instances for each metamorphic relation.

Metamorphic Relation	**#Positive**	**#Negative**
Permutative	34	66
Additive	57	43
Multiplicative	68	32
Invertive	65	35
Inclusive	33	67
Exclusive	31	69

SVM as well as the parameters of the graph kernels. In nested cross validation, values for parameters are selected by performing cross validation on training examples of each fold. We used the SVM implementation in the PyML Toolkit[6] in this work. We used BSR and AUC as evaluation measures when reporting our results.

4.4. RESULTS

We present the results of our empirical studies in this section. We first evaluated the effectiveness of the two graph kernels used in this study. We specifically looked at how the effectiveness of these kernels vary with the values given for the parameters in these two kernels. Then we compared the effectiveness of the graph kernels with the features used in our previous work. Finally we compared the effectiveness of control flow and data dependency information for predicting MRs.

[6]http://pyml.sourceforge.net/

4.4.1. Effectiveness of the random walk kernel. With the random walk kernel, we first evaluated how the prediction accuracy changes with the weighing parameter λ in the kernel (see Appendix B.1 Equation 14 for more details about λ). For this evaluation we computed the random walk kernel values using only the control flow edges in the graph. We varied the value of λ from 0.1 to 0.9 and evaluated the performance using 10 fold stratified cross validation. We used the kernel normalization described in Appendix B.3 to normalize the random walk kernel. We selected the regularization parameter of the SVM using nested cross validation. Figure 4.5 shows the variation of BSR and AUC with λ for each MR. For some MRs, such as additive and multiplicative MRs, there was a considerable variation in accuracy with λ. For these two MRs, higher values of λ gave better performance than lower values of λ. Since each walk of length n is weighted by λ^n, with higher λ values, the random walk kernel value will have a higher contribution from longer walks (see Equation (14) in Appendix B.1). Therefore, for predicting additive and multiplicative MRs, the contribution from long walks in the CFGs seems to be important. For the other four MRs the accuracy did not vary considerably with λ. These results show that the length of random walks has an impact on the prediction effectiveness of some MRs.

Next, we evaluated the effects of the modifications that we made to the node kernel used with the random walk kernel. For this we created separate prediction models using k_{node^1} and k_{node^2} described in equation 10 and equation 11 in Appendix B.1. We used nested cross validation to select the regularization parameter of the SVM and the λ parameter of the random walk kernel. Figure 4.6 shows the performance comparison between k_{node^1} and k_{node^2}. Figure 4.6b shows that the modified node kernel k_{node^2} improves the prediction effectiveness of all the metamorphic relations. Therefore in what follows we use k_{node^2}.

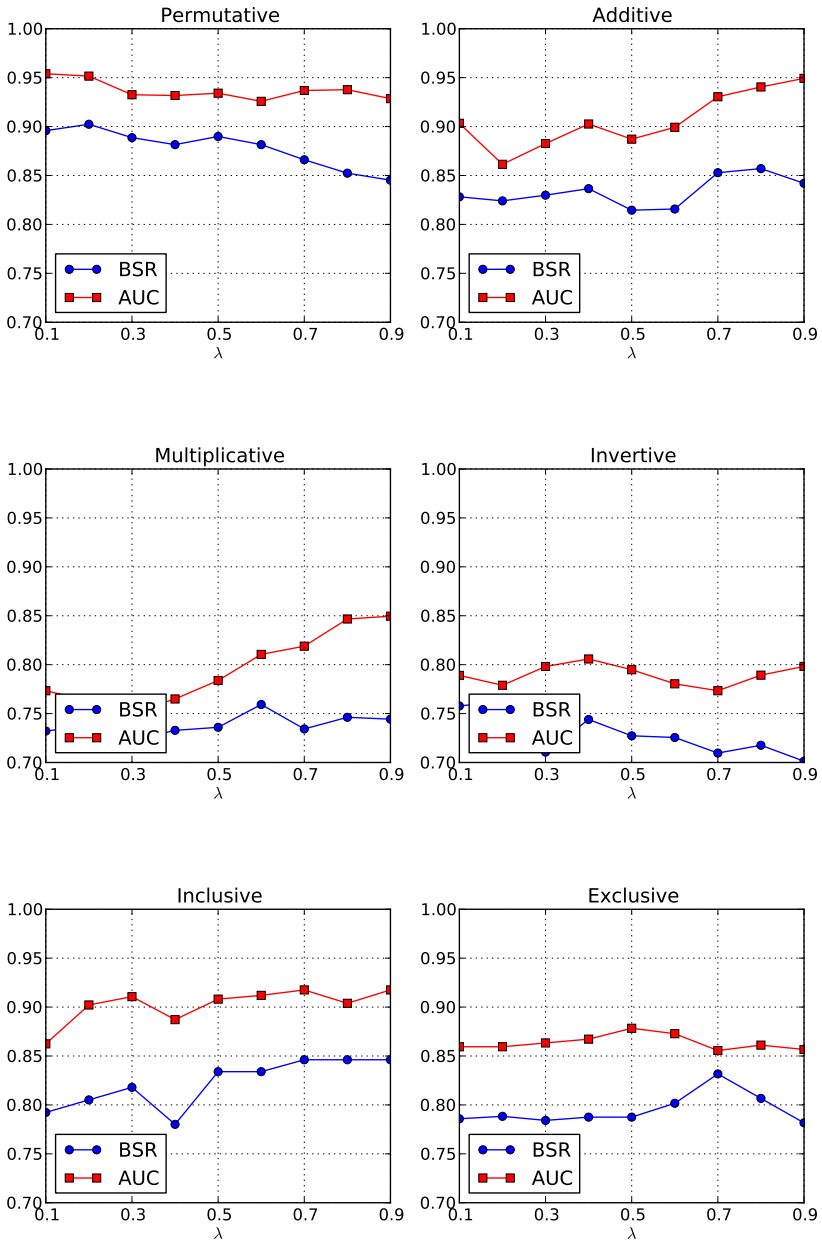
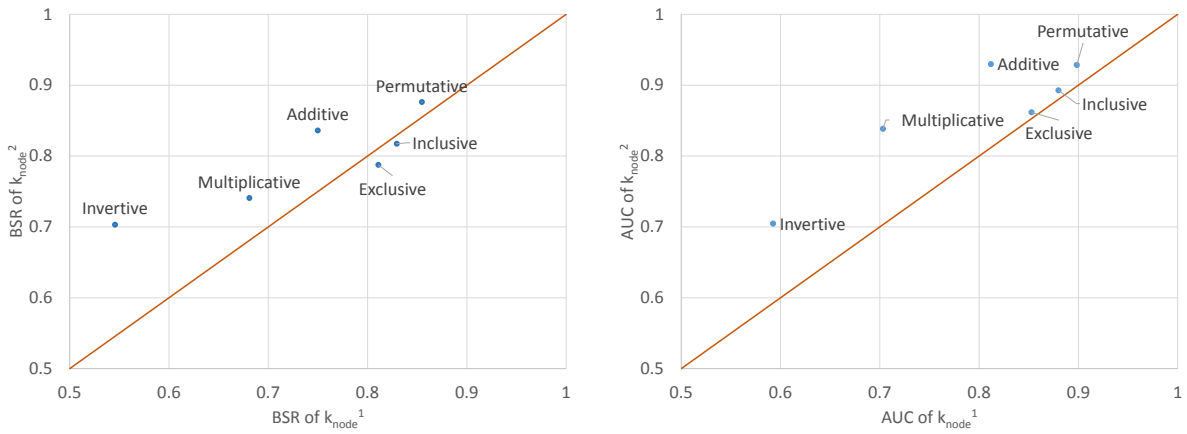


FIGURE 4.5. Variation of the BSR and the AUC with the parameter λ in the random walk graph kernel for each MR.

To identify whether adding more functions to our code corpus would help to increase the accuracy, we plotted learning curves for the random walk kernel. A learning curve shows how the prediction accuracy varies with the training set size. Figure 4.7 shows the learning curves for the six MRs. To generate these curves we held 10% of the functions in our code corpus as the test set. From the other functions we selected subsets 10% to 100% as the



(A) Comparison of BSR (B) Comparison of AUC

FIGURE 4.6. Performance comparison of k_{node^1} and k_{node^2} for the random walk kernel.

training set. We repeated this process 10 times so that the test set would have a different set of functions each time. For all the MRs the AUC values increased as the training set size increases. But the AUC values did not converge indicating that adding more training instances might improve the prediction accuracy further.

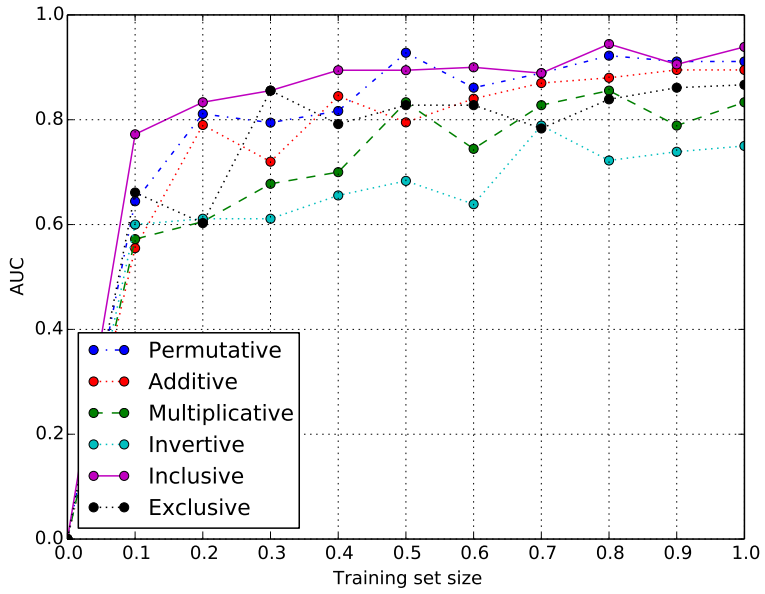
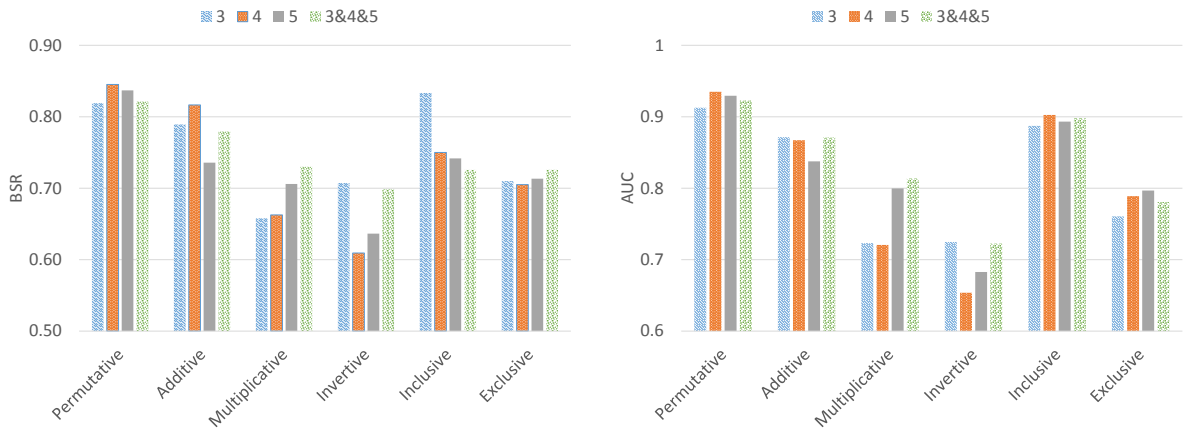


FIGURE 4.7. Learning curves for the six metamorphic relations

4.4.2. Effectiveness of the graphlet kernel. With the graphlet kernel, we first evaluated how prediction accuracy varies with graphlet size. For this evaluation, we used only the control flow edges in the graphs when computing the graphlet kernel values. We used cosine normalization described in Appendix B.3 to normalize the random walk kernel. We varied the graphlet size from three nodes to five nodes and created separate predictive models. In addition, we used all the graphlets together and created a single predictive model. Figure 4.8a and Figure 4.8b shows how the average BSR and average AUC varies with the graphlet size for each MR. When predicting the multiplicative MR, the predictive model created using graphlets of size 5 performed the best. For the invertive MR, graphlets with three nodes performed better than the other predictive models. For the other MRs, all the predictive models gave a similar performance.



(a) BSR (b) AUC

Figure 4.8. Variation of performance with the graphlet size. 3, 4 and 5 are the predictive models created using graphlets of size 3, 4 and 5 respectively. 3&4&5 is the performance of the predictive model created using all the graphlets.

Similar to the random walk kernel, we also compared the performance of the graphlet kernel with the node kernels k_{node^1} and k_{node^2}. As with the random walk kernel, the graphlet kernel with k_{node^2} performed better than the graphlet kernel with k_{node^1}.

4.4.3. Comparison of feature extraction methods. We compared the performance of the node/path features that we used in our initial study [69] with the two graph kernels used in this study. Below we present the details of these three feature extraction methods used for this evaluation:

(1) Node/path features: we followed the same protocol as our earlier work [69]. Node/-path features were calculated using only the "cfg" edges of the graphs. Node features are created by combining the operation performed in the node, its in-degree and its out-degree. Path features are created by taking the sequence of nodes in the shortest path from N_{start} to each node and the sequence of nodes in the shortest path from each node N_{exit}. We used the linear kernel over these features as it exhibited the best performance in our previous experiments [69].

(2) Random walk kernel: we computed the random walk kernel using only the "cfg" edges of the graphs.

(3) Graphlet kernel: we computed the graphlet kernel using only the "cfg" edges of the graphs.

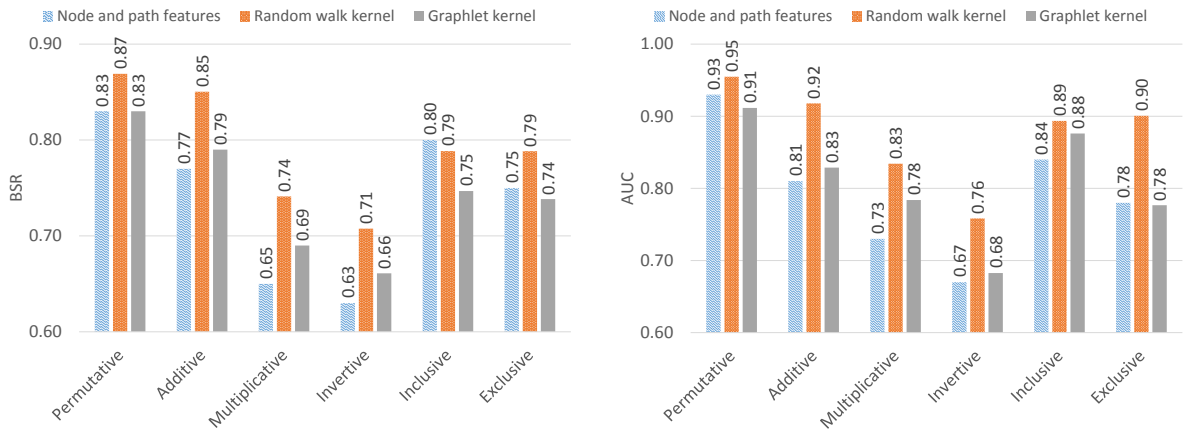
Figure 4.9a and Figure 4.9b shows the average BSR and average AUC for the three feature extraction approaches: node and path features, random walk kernel and graphlet kernel. Among the three feature extraction approaches, the random walk kernel gave the best prediction accuracy for all the MRs. Except for the permutative MR, the AUC values of the graphlet kernel were equal or greater than the AUC values of node/path features. This

improvement in performance could be attributed to the flexibility provided by the graph kernels. The node kernel can be used to compare high level properties of the operations such as commutativity of mathematical operations, in addition to direct comparison of node labels. In fact, Figure 4.6 shows that using such high level properties of operations would improve the accuracy of the prediction models.

Our results show that the random walk kernel performs better than the graphlet kernel for most of the MRs. The random walk kernel compares two control flow graphs based on their walks. A walk in the control flow graph corresponds to a potential execution trace of the function. Therefore it computes a similarity score between two programs based on potential execution traces. The graphlet kernel compares two control flow graphs based on small subgraphs. These subgraphs will capture decision points in the program such as if-conditions. A metamorphic relation is a relationship between outputs produced by multiple executions of the function. Some execution traces directly correlate with some metamorphic relations. Therefore the random walk kernel, which uses execution traces to compare two functions should perform better than the graphlet kernel.

We also evaluated the effect of adding the random walk kernel and the graphlet kernel. Adding the two kernel values is equivalent to using both graph substructures, walks and subgraphs for creating a single model. We added the random walk kernel value and the graphlet kernel value computed for a pair of functions and created a single kernel matrix with these added values. Our experiments showed that this combined kernel could not outperform the random walk kernel. Therefore we do not discuss these results in detail.

4.4.4. Effectiveness of control flow and data dependency information. Finally, we compared the effectiveness of the control flow and data dependency information



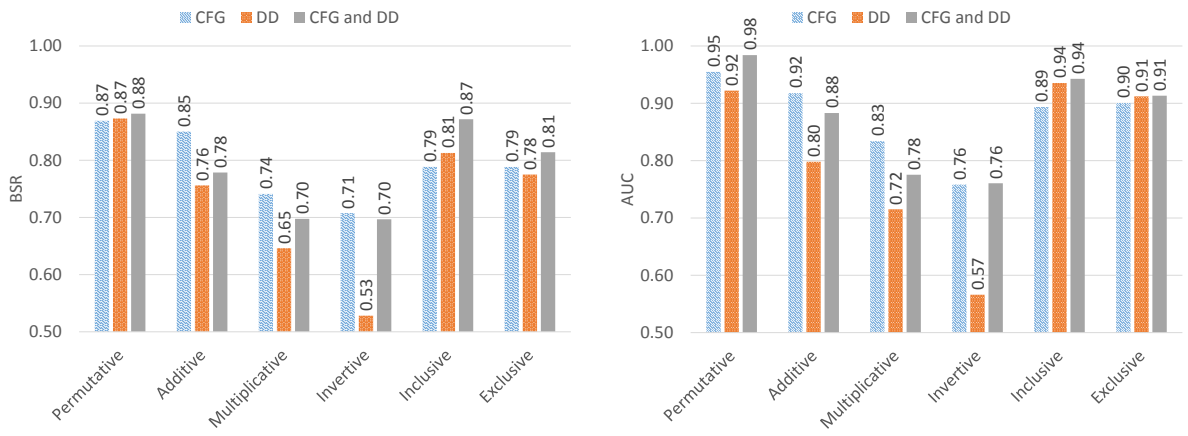
(A) BSR

(B) AUC

FIGURE 4.9. Prediction accuracy of node and path features, random walk kernel and graphlet kernel.

of a function for predicting metamorphic relations. Figure 4.10 shows the effectiveness of using only control flow information, only data dependency information and both control flow and data dependency information. Since the random walk kernel performed best, we used it for this analysis. For this experiment we computed the random walk kernel values separately using the following edges in the graphs: (1) only "cfg" edges, (2) only "dd" edges and (3) both "cfg" and "dd" edges.

We observe that overall, "cfg" edges performed much better than "dd" edges. Performance using "dd" edges was particularly low for the invertive MR. This is because the control flow graph more directly represents information about the execution of the program compared to the data dependency edges. Typically, combining informative features improves classifier performance. We observed this effect in four out of the six MRs. The reduced performance for the multiplicative MR might be the result of the poor performance of the "dd" edges.



(A) BSR (B) AUC

FIGURE 4.10. Performance of control flow and data dependency information using random walk graph kernel. CFG - using only CFG edges, DD - using only data dependency edges, CFG and DD - using both CFG and data dependency edges.

4.5. SAXS: A CASE STUDY

We conducted a case study to evaluate the effectiveness of predicted metamorphic relations in detecting faults in a real world scientific software. For this we used the SAXS [7] program, which analyzes small angle x-ray scattering data. The SAXS program reconstructs macromolecular structures using scattering patterns obtained from experiments. For this case study we used the following three functions that performed the main calculations used in the program:

- *calculateDistance*: computes distance between atoms
- *findGyrationRadius*: computes gyration radius of groups of atoms
- *scatterSample*: main function responsible for scattering

We created faulty versions of these three functions using the μJava[8] mutation engine. We used all the traditional mutation operators provided by μJava for generating the mutants.

[7] `http://www.cs.colostate.edu/hpc/SAXS/index.php`
[8] `http://cs.gmu.edu/~offutt/mujava/`

Table 4.4 shows the number of mutants that we used in the case study after removing the mutants that gave exceptions or infinite loops. We also removed mutants that gave the same outputs as the original version of the function for the test cases used in the experiment (listed as equivalent mutants in the table).

TABLE 4.4. Details of mutants of SAXS functions

	calculateDistance	findGyrationRadius	scatterSample	Total
Exceptions	15	62	33	110
Infinite loops	1	1	1	3
Equivalent mutants	4	19	12	35
No. of mutants used	19	54	139	212

Initially we obtained metamorphic relation predictions for the original SAXS functions using the the graph kernel based metamorphic relation system described in Section 4.1. Then, for each of these faulty versions of the functions we obtained metamorphic relation predictions using the same approach. For training the machine learning classification models we used all of the functions in the code corpus described in Section 4.3.2.

We randomly generated 10 initial test cases for each of the functions. Using the predicted metamorphic relations we created the corresponding follow-up test cases. Finally we executed these test case pairs on the functions and checked whether the predicted metamorphic relations were violated. Violation of a predicted metamorphic relation indicates that the mutant was detected through metamorphic testing. For the original version of the SAXS functions we could not observe any violations of the predicted metamorphic relations. Table 4.5 shows the percentage of mutants killed using the predicted MRs. From all the mutants of the three functions, 90% of the mutants could be killed using the predicted MRs. This shows that the MRs predicted by our method can be effective in identifying faults in real world scientific programs.

TABLE 4.5. Mutants detected by predicted MRs. f_1: *calculateDistance*, f_2: *findGyrationRadius*, and f_3: *scatterSample*

	f_1	f_2	f_3	Total
No. of mutants used	19	54	139	212
Killed by MT (using predicted MRs)	19	45	127	191
Percentage of killed mutants (%)	100	83.33	91.37	90.09

CHAPTER 5

Effectiveness of Multi-label Classification for Metamorphic Relation Prediction

In this Chapter we investigate the effectiveness of multi-label classification for predicting metamorphic relations. We compare the effectiveness of different multi-label classification algorithms. We also compare these results with the results we obtained from binary classifiers presented in the previous chapters.

5.1. Method overview

Figure 5.1 shows an overview of the method used for applying multi-label classification for predicting metamorphic relations. During the training phase, we start by extracting the node/path features described in Section 3.2.2 from the functions in the training set. Training data set in Figure 5.1 shows how multiple labels are associated with each function in the data set corresponding to each metamorphic relation satisfied by that function. Then this training data set is used by a multi-label classification algorithm to create a prediction model.

During the testing phase the same set of features are extracted from the functions in order to get predictions of metamorphic relations. Then these features are supplied to the previously trained prediction model. The prediction model uses these features to provide predictions for the set of metamorphic relations satisfied by each function.

5.1.1. Multi-label classification algorithms. In this work we used four multi-label classification algorithms. Figure 5.2 shows the classification of multi-label algorithms used in this work. More details about these algorithms can be found in Section 2.3.2. Two

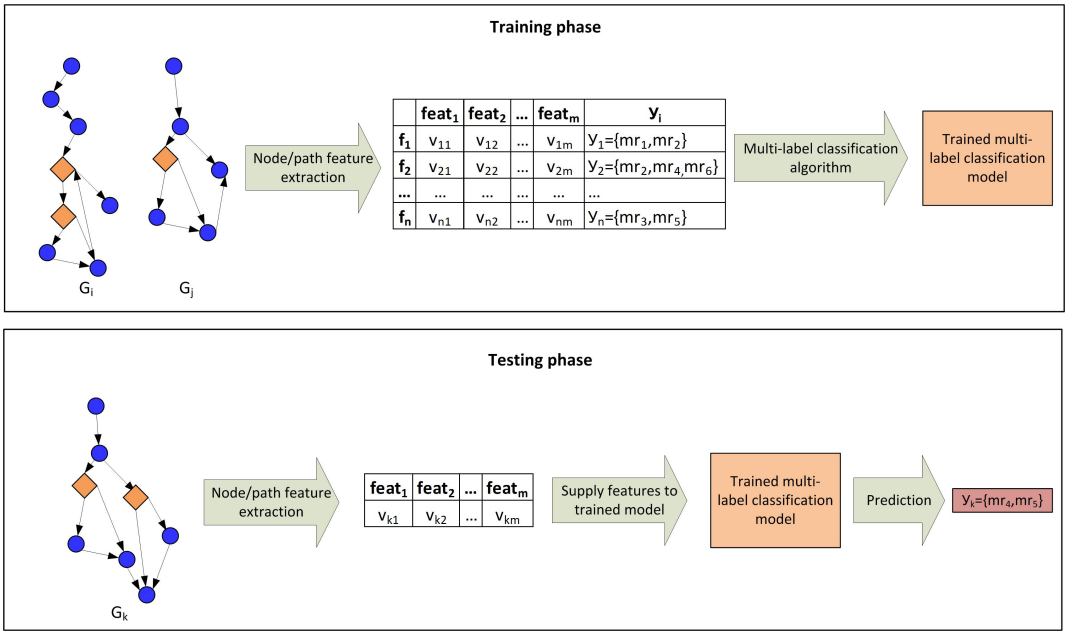


FIGURE 5.1. Overview of the multi-label classification method

of these algorithms fall into the category of problem transformation methods: (1) calibrated label ranking (CLR) [59] and (2) hierarchy of multi-label classifiers (HOMER) [60]. The next two approaches, multi-label k-nearest neighbors (ML-kNN) [63] and BP-MLL [65] are algorithm adaptation methods. The final two algorithms, RAkEL [61] and ECC [62] are ensemble approaches. As described in Section 2.3.2, both problem transformation methods and ensemble methods use binary classifiers as base classifiers. In this work we used SVMs and decision trees as base classifiers for CLR, HOMER, ECC,and RAkEL.

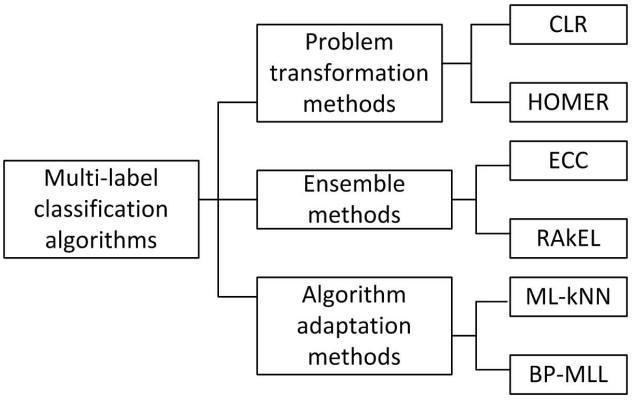


FIGURE 5.2. Classification of multi-label algorithms used

5.1.2. Experimental Setup. For this evaluation we used the same code corpus described in Section 4.3.2. We extracted the node/path features described in Section 3.2.2 from the graph representations of these functions. Similar to the experiments done with binary classifiers we focused on predicting the six metamorphic relations listed in Table 4.2. Therefore these are the six labels in our multi-label classification problem.

When conducting the experiment, we used the implementations of the above mentioned four multi-label classification algorithms provided in the MULAN[1] library. When using the SVM as the base classifier, we used the linear kernel and set the regularization parameter C to 10 since these parameter settings gave good results in our previous experiments. As described in Section 2.3.2.3, for RAkEL we had to select two parameters: number of base classifiers and size of the label subset. We set the number of base classifiers to 12 (2× number of labels) and the size of the label subsets to 3 (number of labels/2). Tsoumakas et al. [61] has shown that these are reasonable choices for these parameters.

To obtain the results we used 10-fold cross validation as with our previous experiments. Then we calculated the evaluation measures described in Section 2.3.4 to measure the effectiveness of the multi-label classification algorithms.

5.2. Results

5.2.1. Performance of problem transformation methods. First we compare the performance of the two problem transformation methods: CLR and HOMER. Table 5.1 shows the results obtained using these three algorithms. Within parentheses we present the standard deviation of the value for each measure. Values highlighted in bold shows the best performance for each evaluation measure. CLR-SVM and CLR-DT refers to using SVMs

[1]http://mulan.sourceforge.net/

and decision trees as base classifiers for the CLR algorithm. Similarly HOMER-SVM and HOMER-DT refers respectively to using SVMs and decision trees as base algorithms.

TABLE 5.1. Effectiveness of problem transformation methods for metamorphic relation prediction

	CLR-SVM	CLR-DT	HOMER-SVM	HOMER-DT
Micro-precision	**0.72** (0.13)	0.67 (0.11)	0.68 (0.13)	0.70 (0.13)
Micro-recall	0.81 (0.10)	0.77 (0.15)	**0.84** (0.12)	0.71 (0.13)
Micro-F	**0.75** (0.07)	0.70 (0.10)	0.74 (0.08)	0.69 (0.08)
Macro-precision	**0.71** (0.15)	0.67 (0.12)	0.70 (0.14)	0.69 (0.12)
Macro-recall	0.77 (0.09)	0.76 (0.13)	**0.84** (0.12)	0.69 (0.14)
Macro-F	0.71 (0.09)	0.68 (0.09)	**0.72** (0.07)	0.65 (0.09)

CLR-SVM outperformed CLR-DT when considering all of the six performance measures. HOMER-SVM outperformed HOMER-DT for five evaluation measures. This shows that when using problem transformation methods for predicting metamorphic relations, using SVMs as base classifiers can improve the performance over the use of decision trees. This is consistent with the performance we observed with binary classifiers as shown in Section 3.3.3.2.

When comparing the two problem transformation methods, the HOMER algorithm outperformed CLR when considering both macro-recall and micro-recall measures. This means that HOMER predicted more metamorphic relations than CLR at the cost of predicting false positives. Therefore it might be useful to use HOMER for predicting MRs especially when testing safety critical systems. For safety critical systems conducting more testing with more MRs may be more important than having several false positives. On the other hand, CLR outperformed HOMER when considering both micro-precision and macro-precision. Therefore if you are conducting testing with a limited testing budget it might be better to use CLR to get the predictions since it will predict fewer incorrect MRs.

5.2.2. Performance of ensemble methods. Next, we evaluated the effectiveness of ensemble methods for predicting MRs. We used two ensemble methods: RAkEL and ECC described in Section 2.3.2.3. Table 5.2 displays the results obtained using these two ensemble methods. RAkEL-SVM and RAkEL-DT refers to using SVMs and decision trees as base classifiers with the RAkEL algorithm. Similarly, ECC-SVM and ECC-DT refers to using SVMs and decision trees as base classifiers with the ECC algorithm.

Table 5.2. Effectiveness of ensemble methods for predicting metamorphic relations

	RAkEL-SVM	RAKEL-DT	ECC-SVM	ECC-DT
Micro-precision	0.69 (0.14)	**0.71** (0.15)	0.70 (0.13)	0.69 (0.12)
Micro-recall	**0.80** (0.12)	0.67 (0.12)	**0.80** (0.08)	0.71 (0.13)
Micro-F	**0.73** (0.09)	0.67 (0.09)	**0.73** (0.08)	0.68 (0.07)
Macro-precision	0.70 (0.17)	0.70 (0.17)	**0.71** (0.17)	0.68 (0.14)
Macro-recall	**0.78** (0.14)	0.65 (0.12)	0.75 (0.08)	0.70 (0.13)
Macro-F	**0.69** (0.10)	0.63 (0.09)	**0.69** (0.09)	0.65 (0.08)

RAkEL-DT outperformed RAkEL-SVM when considering micro-precision and both of them performed equally when considering the macro-precision measure. But for the other evaluation measures, RAkEL-SVM outperformed RAkEL-DT. ECC-SVM outperformed ECC-DT when considering all the evaluation measures. Therefore, similar to problem transformation methods, ensemble methods also performed better when using SVMs as base classifiers.

When comparing the two ensemble methods, RAkEL and ECC, there does not seem to be a clear winner. RAkEL outperformed ECC when considering the macro-recall measure. All the other measures gave close values for the two algorithms.

5.2.3. Performance of algorithm adaptation methods. Finally, we evaluated the performance of two algorithm adaptation methods, ML-kNN and BP-MLL described in Section 2.3.2.4 and Section 2.3.2.5, respectively. Table 5.3 shows the the results obtained

using the two algorithm adaptation methods. For three evaluation measures, BP-MLL outperformed ML-kNN. But the standard deviations reported by BP-MLL were considerably high compared to the standard deviations reported by ML-kNN.

TABLE 5.3. Effectiveness of algorithm adaptation methods for predicting metamorphic relations

	ML-kNN	BP-MLL
Micro-precision	**0.63** (0.11)	0.46 (0.18)
Micro-recall	0.63 (0.11)	**0.77** (0.30)
Micro-F	**0.62** (0.09)	0.57 (0.21)
Macro-precision	**0.48** (0.15)	0.37 (0.16)
Macro-recall	0.52 (0.09)	**0.72** (0.30)
Macro-F	0.46 (0.09)	**0.47** (0.19)

5.2.4. PERFORMANCE COMPARISON ACROSS DIFFERENT MULTI-LABEL CLASSIFICATION METHODS. Table 5.4 shows the performance comparison of the three categories of multi-label classification algorithms used in this work. Problem transformation algorithms performed best when considering the six evaluation measures. The algorithm adaptation methods exhibited the worst performance.

TABLE 5.4. Performance comparison of multi-label classification methods

	Problem transformation methods	Ensemble methods	Algorithm adaptation methods
Micro-precision	**0.72**	0.71	0.63
Micro-recall	**0.84**	0.80	0.63
Micro-F	**0.75**	0.73	0.62
Macro-precision	**0.71**	**0.71**	0.48
Macro-recall	**0.84**	0.78	0.52
Macro-F	**0.72**	0.69	0.46

5.2.5. PERFORMANCE COMPARISON OF MULTI-LABEL CLASSIFIERS AND BINARY CLASSIFIERS. To compare the performance of multi-label classifiers with binary classifiers, we calculated the same set performance measures as done for the multi-label classifiers using

the results obtained with binary classifiers. We created the binary classifiers using SVMs with the linear kernel and regularization parameter set to 10. We used the same set of node/path features to train the SVMs that we used to create the multi-label classifiers. Even though the random walk kernel performed better than node/path features, we did not use graph kernels in this evaluation, since implementations of most of the multi-label classification algorithms used in this study do not accommodate using graph kernels with them.

Table 5.5 shows the comparison of the two classification methods. Binary classification outperformed multi-label classification when considering both the micro-precision and macro-precision measures. For the micro-recall and macro-recall measures, multi-label classification algorithms performed better than binary classification algorithms. Multi-label classification algorithms consider dependencies among labels. Thus considering label dependencies may help to improve the recall of the classifier but not its precision.

TABLE 5.5. Performance comparison of multi-label classification algorithms with binary classification algorithms

	Binary classifiers	Multi-label classifiers
Macro-precision	**0.82**	0.72
Macro-recall	0.74	**0.84**
Macro-F	**0.78**	0.75
Micro-precision	**0.83**	0.71
Micro-recall	0.75	**0.84**
Micro-F	**0.79**	0.72

5.3. CONCLUSIONS

In this Chapter we evaluated the effectiveness of multi-label classification algorithms for predicting metamorphic relations. Specifically we investigated whether the use of dependencies among labels by the classification algorithm will improve prediction accuracy. Our

results show that including dependencies among metamorphic relations can increase the classifier recall. In addition, problem transformation algorithms gave the best performance in multi-label classification when using SVMs as their base classifiers. We evaluated the performance of two problem transformation algorithms: CLR and HOMER. CLR had better precision than HOMER, while HOMER had better recall than CLR. Therefore the selection of one of these two algorithms should be done depending on the type of program that you plan to test. For example, when testing a safety critical system, using HOMER to get metamorphic relations predictions might be useful, since it recalls most of the metamorphic relations that should be satisfied by the program, when compared to CLR. If your are testing a program with a limited testing budget, CLR maybe more suitable for making metamorphic relation predictions.

CHAPTER 6

Limitations and Threats to Validity

6.1. Limitations

One of the main limitations of our work is that the techniques we present here can only be used to predict metamorphic relations at the function level. So the metamorphic relations predicted by our techniques can be used as automated test oracles for conducting automated unit testing. We believe that predicting metamorphic relations for a whole program would be a very difficult task due to high variability among programs. Further, previous studies have shown that higher number of metamorphic relations can be usually derived at the function level compared to deriving metamorphic relations for a whole program [88]. Also, previous work has shown that function level metamorphic relations can reveal defects that were not found by metamorphic relations derived for the whole program [88]. Therefore even if the techniques that we present here can only predict metamorphic relations at the function level they can be highly effective in identifying faults when conducting unit testing.

Another limitation of our work is that we used only numerical programs that handle numerical inputs/outputs and a set of metamorphic relations that are commonly found in such programs in our evaluations. But we believe that our approach will extend to programs that handle other types of inputs/outputs such as graphs, matrices, etc. In such cases prediction models need to be trained for predicting metamorphic relations that are commonly found in such programs.

6.2. Threats to validity

6.2.1. External validity. Conditions that limit the ability to generalize experimental results are considered as threats to external validity [89]. We used a code corpus consisting of 100 mathematical functions for our empirical studies. These functions differ in size and complexity. They also perform different functionalities such as sorting, searching, calculating common statistics, etc. But this set of functions can pose a threat to external validity. We tried to minimize this threat by using functions obtained from different open source libraries. Even though we only used a set of 100 mathematical functions, the results demonstrate the effectiveness of this novel approach for detecting likely metamorphic relations.

6.2.2. Internal validity. Influences that can effect the independent variable of an experiment with respect to causality are called internal validity threats [89]. Threats to internal validity can occur due to potential faults in the implementations of the functions. Since we were not the developers of these functions we cannot guarantee that these functions are free of faults. The competent programmer hypothesis [90] states that even though the programmer might make some mistakes in the implementation, the general structure of the program should be similar to the fault-free version of the program. We use control flow and data dependency information about a program to create our prediction models. According to the competent programmer hypothesis this information should not change significantly even with a fault. In addition, there may be more relevant metamorphic relations for these functions than the six metamorphic relations that we used for our empirical studies.

6.2.3. Conclusion validity. Issues that affect the ability to draw correct conclusions about the relationships between treatments and outcomes in experiments are called threats to conclusion validity [89]. The main threat to conclusion validity is the sample size used

in the validation. We used 100 programs that take numerical inputs and produce numerical outputs in this study. We limited the set of programs to 100 since we believe it is not cost effective for the classifier to learn from a larger set of programs.

6.2.4. Construct validity. Any concerns related to generalizing the results of an experiment to the theory behind the experiment are considered as threats to construct validity [89]. We used the *Soot* framework to generate CFGs of the functions used in this experiment. Further we used the *NetworkX*[1] package for graph manipulation. Use of these third party tools represents potential threats to construct validity. We verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool. Further, we used mutation analysis to measure the fault detection effectiveness of predicted metamorphic relations. Mutation analysis represents a threat to construct validity because mutations are synthetic faults. However, previous studies have shown that mutations represent faults made by a real human programmer [76].

[1]http://networkx.lanl.gov/

CHAPTER 7

Conclusion

In this chapter we describe our contributions, possible future work, and conclusions.

7.1. Contributions

In this dissertation we introduced a novel machine learning based approach for predicting metamorphic relations in previously unseen functions. These predicted metamorphic relations can be used as test oracles during automated testing. We evaluated the different components of our approach: machine learning algorithms, feature extraction techniques and the program information used to develop the features. Our key contributions include the following:

- We are the first to present a systematic literature survey on testing scientific software (Section 2.1). Through this systematic literature survey we found that challenges faced when testing scientific software fall into two main categories: (1) testing challenges that are due to characteristics of scientific software such as oracle problems and (2) testing challenges that are caused by cultural differences between scientists and the software engineering community, for example viewing the code and the model that it implements as inseparable entities. In addition, we identified methods to potentially overcome these challenges and their limitations. Finally we described unsolved challenges and how software engineering researchers and practitioners can help to overcome them.
- We presented a novel method for predicting metamorphic relations at the function level (Chapter 3). We developed a set of features using function's control flow graphs

for creating binary classification models. Then we evaluated the effectiveness of three binary classification algorithms using a set numerical functions widely used in scientific computing. We found that support vector machines are the most effective of the three in predicting metamorphic relations.

- We conducted empirical evaluations of the effectiveness of different types of features and program information used to develop the machine learning classification models (Chapter 4). We used graph kernels to develop different categories of features that represent semantic information about the programs. To our knowledge this is the first time graph kernels have been developed to capture semantic information in source code. We found that graph kernels developed using walks in the graphs are highly effective in prediction metamorphic relations. Then we conducted empirical evaluations to evaluate the effectiveness of control flow and data dependency information for predicting metamorphic relations. Through this we found that control flow information is more effective in predicting metamorphic relations than data dependency information.
- We evaluated the effectiveness of multi-label classification algorithms for predicting metamorphic relations. Our empirical studies showed that the precision of multi-label classifiers is lower than that of binary classifiers. But multi-label classifiers have a higher recall than binary classifiers. We also showed that problem transformation methods works best for metamorphic relation prediction.

Work in this dissertation has also led to publication of two journal papers ([12] and one currently under review) and three conference/workshop papers [69, 91, 92]. Our work also resulted in an invited contribution to a book chapter [93].

7.2. Future work

Automated test oracles are essential for conducting systematic automated testing. There are a number of interesting future work possibilities for developing such automated oracle. We discuss several of those future work possibilities below:

(1) Automatically deriving program specific metamorphic relations: in this work we focused on predicting whether a function should satisfy a set of metamorphic relations commonly found in numerical programs. For example the additive metamorphic relation specifies that when you add a positive constant to the initial input, the output will increase or remains constant. This metamorphic relation does not specify the constant value or the exact increase in the output, which is specific to the program under test. Such program specific metamorphic relations can be automatically generated using search-based techniques. This analysis will require using dynamic properties of the program.

(2) Hybrid test oracles: Effectiveness of test oracles may be improved by combining automatically generated metamorphic relations with automatically generated program assertions. These hybrid oracles will be especially useful for testing scientific programs that face oracle problems. But these hybrid oracles needs to be created in an effective manner so that the testing costs do not increase.

(3) Test oracle selection: Automatic generation of metamorphic relations will result in generating multiple metamorphic relations for a given program. Then it is important to select or prioritize the most effective oracle for reducing testing costs. Therefore developing a set of metrics to measure the effectiveness of test oracles will be useful. These metrics can be used to evaluate different features offered by different test

oracles for a given program. For example, the metrics can help one to select between multiple metamorphic relations when a testing budget is limited. These measures need to consider different aspects of test oracles such as soundness and completeness in addition to their fault finding ability.

7.3. Conclusion

Testing programs that do not have oracles is a continuing challenge in software testing [94]. In this dissertation we addressed the problem of automatically predicting metamorphic relations that can be used as test oracles for programs that do not have oracles. We developed a novel technique that creates machine learning models using semantic features extracted from graph based program representations. We showed this method can effectively predict metamorphic relations using a set of real world functions. In addition, we showed that these predicted metamorphic relations can detect faults in these functions. Techniques that we developed here reduce the testing cost by minimizing the human involvement required in discovering the metamorphic relations for a given program.

Automatically predicted metamorphic relations can be used as automated test oracles when conducting automated testing especially on scientific software. These predicted metamorphic relations can serve as automated test oracles during the testing process of scientific programs and thus remove the need for a domain expert to be involved during the testing of these programs. This automated process reduces the testing costs as well as the subjectivity of the testing process. Systematic automated testing can help to reveal more faults in scientific programs and increase our confidence in the results produced by these programs.

Bibliography

[1] D. E. Post and R. P. Kendall, "Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from ASCI," *International Journal of High Performance Computing Applications*, vol. 18, pp. 399–416, Nov. 2004.

[2] J. B. Drake, P. W. Jones, and G. R. Carr, Jr., "Overview of the software design of the community climate system model," *International Journal of High Performance Computing Applications*, vol. 19, pp. 177–186, Aug. 2005.

[3] R. Sanders and D. Kelly, "Dealing with risk in scientific software development," *Software, IEEE*, vol. 25, pp. 21 –28, July–Aug. 2008.

[4] L. Hatton, "The T experiments: errors in scientific software," *Computational Science Engineering, IEEE*, vol. 4, pp. 27 –38, Apr.–June 1997.

[5] A. J. Abackerli, P. H. Pereira, and N. Calônego Jr., "A case study on testing CMM uncertainty simulation software (VCMM)," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 32, pp. 8 – 14, Mar. 2010.

[6] G. Miller, "A scientist's nightmare: Software problem leads to five retractions," *Science*, vol. 314, no. 5807, pp. 1856–1857, 2006.

[7] L. Hatton and A. Roberts, "How accurate is scientific software?," *IEEE Transactions on Software Engineering*, vol. 20, pp. 785 –797, Oct. 1994.

[8] P. Dubois, "Testing scientific programs," *Computing in Science Engineering*, vol. 14, pp. 69 –73, July–Aug. 2012.

[9] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[10] R. Sanders and D. Kelly, "The challenge of testing scientific software," in *Proceedings Conference for the Association for Software Testing (CAST)*, pp. 30–36, July 2008.

[11] L. N. Joppa, G. McInerny, R. Harper, L. Salido, K. Takeda, K. O'Hara, D. Gavaghan, and S. Emmott, "Troubling trends in scientific software use," *Science*, vol. 340, no. 6134, pp. 814–815, 2013.

[12] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and Software Technology*, vol. 56, no. 10, pp. 1219 – 1232, 2014.

[13] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Tech. Rep. HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

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

[15] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, "Metamorphic testing and its applications," in *Proceedings of the 8th International Symposium on Future Software Technology*, pp. 345–351, Software Engineers Association, 2004.

[16] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544 – 558, 2011.

[17] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou, "Case studies on the selection of useful relations in metamorphic testing," in *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering*, pp. 569–583, 2004.

[18] S. M. Easterbrook, "Climate change: a grand software challenge," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 99–104, ACM, 2010.

[19] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil, "On effective testing of health care simulation software," in *Proceedings of the 3rd Workshop on Software Engineering in Health Care*, pp. 40–47, 2011.

[20] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Proceedings of the 29th international conference on Software Engineering*, pp. 550–559, IEEE Computer Society, 2007.

[21] D. Kelly, S. Smith, and N. Meng, "Software engineering for scientists," *Computing in Science Engineering*, vol. 13, pp. 7 –11, Sept.–Oct. 2011.

[22] M. T. Sletholt, J. Hannay, D. Pfahl, H. C. Benestad, and H. P. Langtangen, "A literature review of agile practices and their effects in scientific software development," in *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, SECSE '11, (New York, NY, USA), pp. 1–9, ACM, 2011.

[23] C. Murphy, G. E. Kaiser, and M. Arias, "An approach to software testing of machine learning applications," in *Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering*, pp. 167–172, July 2007.

[24] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC Bioinformatics*, vol. 10, pp. 1–12, 2009.

[25] D. Kelly and R. Sanders, "Assessing the quality of scientific software," in *First International Workshop on Software Engineering for Computational Science and Engineering*, 2008.

[26] J. Pitt-Francis, M. O. Bernabeu, J. Cooper, A. Garny, L. Momtahan, J. Osborne, P. Pathmanathan, B. Rodriguez, J. P. Whiteley, and D. J. Gavaghan, "Chaste: using agile programming techniques to develop computational biology software," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 366, no. 1878, pp. 3111–3136, 2008.

[27] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pp. 1–8, IEEE Computer Society, 2009.

[28] J. Segal, "Scientists and software engineers: A tale of two cultures," in *20th Annual Meeting of the Pschology of Programming Interest Group*, Lancaster University, Sept. 2008.

[29] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, "A survey of scientific software development," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 12:1–12:10, ACM, 2010.

[30] M. A. Heroux, J. M. Willenbring, and M. N. Phenow, "Improving the development process for CSE software," in *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*, pp. 11 –17, Feb. 2007.

[31] D. Kelly, R. Gray, and Y. Shao, "Examining random and designed tests to detect code mistakes in scientific software," *Journal of Computational Science*, vol. 2, no. 1, pp. 47 – 56, 2011.

[32] D. Kelly, S. Thorsteinson, and D. Hook, "Scientific software testing: Analysis with four dimensions," *Software, IEEE*, vol. 28, pp. 84 –90, May–June 2011.

[33] T. Clune and R. Rood, "Software testing and verification in climate model development," *Software, IEEE*, vol. 28, pp. 49 –55, Nov.–Dec. 2011.

[34] P. E. Farrell, M. D. Piggott, G. J. Gorman, D. A. Ham, C. R. Wilson, and T. M. Bond, "Automated continuous verification for numerical simulation," *Geoscientific Model Development*, vol. 4, no. 2, pp. 435–449, 2011.

[35] S. M. Easterbrook and T. C. Johns, "Engineering the software for understanding climate change," *Computing in Science Engineering*, vol. 11, pp. 65 –74, nov.-dec. 2009.

[36] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proceedings of the ACM '81 conference*, pp. 254–257, ACM, 1981.

[37] T. Chen, J. Feng, and T. H. Tse, "Metamorphic testing of programs on partial differential equations: a case study," in *Proceedings of the 26th Annual International Computer Software and Applications Conference*, pp. 327–333, 2002.

[38] S. Brilliant, J. Knight, and N. Leveson, "Analysis of faults in an n-version software experiment," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 238–247, 1990.

[39] P. C. Lane and F. Gobet, "A theory-driven testing methodology for developing scientific software," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 24, no. 4, pp. 421–456, 2012.

[40] D. Hook and D. Kelly, "Testing for trustworthiness in scientific software," in *ICSE Workshop on Software Engineering for Computational Science and Engineering*, pp. 59 –64, May 2009.

[41] L. Hochstein and V. Basili, "The ASC-Alliance projects: A case study of large-scale parallel scientific code development," *Computer*, vol. 41, pp. 50–58, Mar. 2008.

[42] J. Mayer, "On testing image processing applications with statistical methods," in *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*, pp. 69–78, 2005.

[43] M. Cox and P. Harris, "Design and use of reference data sets for testing scientific software," *Analytica Chimica Acta*, vol. 380, no. 23, pp. 339 – 351, 1999.

[44] J. Ding, T. Wu, D. Wu, J. Q. Lu, and X.-H. Hu, "Metamorphic testing of a Monte Carlo modeling program," in *Proceedings of the 6th International Workshop on Automation of Software Test*, pp. 1–7, ACM, 2011.

[45] R. Guderlei and J. Mayer, "Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing," in *Proceedings of the Seventh International Conference on Quality Software*, pp. 404 –409, Oct. 2007.

[46] T. Y. Chen, J. Feng, and T. H. Tse, "Metamorphic testing of programs on partial differential equations: A case study," in *Proceedings of the 26th International Computer*

Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, pp. 327–333, IEEE Computer Society, 2002.

[47] C. Murphy, K. Shen, and G. Kaiser, "Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pp. 436–445, IEEE Computer Society, 2009.

[48] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 189–200, ACM, 2009.

[49] D. Zhang and J. J. Tsai, *Advances in Machine Learning Applications in Software engineering*. Idea Group Publishing, 2007.

[50] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, Mar. 1986.

[51] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[52] T. Kohonen, "An introduction to neural computing," *Neural Networks*, vol. 1, no. 1, pp. 3 – 16, 1988.

[53] J. R. Quinlan, *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[54] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explorations Newsletter*, vol. 11, pp. 10–18, Nov. 2009.

[55] H. Zhu, "On information and sufficiency," *Annals of Statistics*, vol. 22, no. 1, pp. 79–86, 1997.

[56] G. Madjarov, D. Kocev, D. Gjorgjevikj, and S. Deroski, "An extensive experimental comparison of methods for multi-label learning," *Pattern Recognition*, vol. 45, no. 9, pp. 3084 – 3104, 2012.

[57] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining*, vol. 2007, pp. 1–13, 2007.

[58] J. Fürnkranz, "Round robin classification," *J. Mach. Learn. Res.*, vol. 2, pp. 721–747, Mar. 2002.

[59] J. Fürnkranz, E. Hüllermeier, E. LozaMencía, and K. Brinker, "Multilabel classification via calibrated label ranking," *Machine Learning*, vol. 73, no. 2, pp. 133–153, 2008.

[60] G. Tsoumakas, I. Katakis, and I. P. Vlahavas, "Effective and Efficient Multilabel Classification in Domains with Large Number of Labels," in *2008 Workshop on Mining Multidimensional Data*, Lecture Notes in Computer Science, pp. 30–44, 2008.

[61] G. Tsoumakas and I. Vlahavas, "Random k-labelsets: An ensemble method for multilabel classification," in *Machine Learning* (J. Kok, J. Koronacki, R. Mantaras, S. Matwin, D. Mladenič, and A. Skowron, eds.), vol. 4701, pp. 406–417, Springer Berlin Heidelberg, 2007.

[62] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier chains for multi-label classification," in *Machine Learning and Knowledge Discovery in Databases* (W. Buntine, M. Grobelnik, D. Mladeni, and J. Shawe-Taylor, eds.), vol. 5782 of *Lecture Notes in Computer Science*, pp. 254–269, Springer Berlin Heidelberg, 2009.

[63] M.-L. Zhang and Z.-H. Zhou, "Ml-knn: A lazy learning approach to multi-label learning," *Pattern Recognition*, vol. 40, no. 7, pp. 2038 – 2048, 2007.

[64] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, pp. 21–27, Sept. 2006.

[65] Z. Z. Zhang, M.L., "Multi-label neural networks with applications to functional genomics and text categorization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, pp. 1338–1351, 2006.

[66] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error-propagation," in *Parallel Distributed Processing* (D. E. Rumelhart and R. J. McClelland, eds.), ch. 8, MIT Press, 1986.

[67] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. New York, NY, USA: Cambridge University Press, 2004.

[68] N. Shervashidze, S. V. N. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient graphlet kernels for large graph comparison," in *Proceedings of the International Workshop on Artificial Intelligence and Statistics. Society for Artificial Intelligence and Statistics*, pp. 488–495, 2009.

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

[70] K. M. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs," in *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005)*, pp. 74–81, IEEE Computer Society, 2005.

[71] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," in *Learning Theory and Kernel Machines* (B. Schölkopf and M. Warmuth, eds.), vol. 2777 of *Lecture Notes in Computer Science*, pp. 129–143, Springer Berlin Heidelberg, 2003.

[72] J. Ramon and T. Grtner, "Expressivity versus efficiency of graph kernels," in *Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences*, pp. 65–74, 2003.

[73] A. Ben-Hur and J. Weston, "A User's Guide to Support Vector Machines," in *Data Mining Techniques for the Life Sciences* (O. Carugo and F. Eisenhaber, eds.), vol. 609 of *Methods in Molecular Biology*, ch. 13, pp. 223–239, Humana Press, 2010.

[74] J. Huang and C. Ling, "Using AUC and accuracy in evaluating learning algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, pp. 299 – 310, Mar. 2005.

[75] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java bytecode for analyses and transformations," Tech. Rep. 1998-4, Sable Research Group, McGill University, July 1998.

[76] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 402–411, ACM, 2005.

[77] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, 2011.

[78] Y. S. Ma and J. Offutt, "Description of method-level mutation operators for java." `http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`, Nov. 2005. Accessed: 03-11-2015.

[79] R. Burbidge and B. Buxton, "An introduction to support vector machines for data mining," pp. 3–15, 2001.

[80] S. Beiden, M. Maloof, and R. Wagner, "A general model for finite-sample effects in training and testing of competing classifiers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, pp. 1561–1569, Dec. 2003.

[81] F. E. Allen, "Control flow analysis," *SIGPLAN Notices*, vol. 5, pp. 1–19, July 1970.

[82] R. I. Kondor and J. Lafferty, "Diffusion kernels on graphs and other discrete structures," in *In Proceedings of the ICML*, pp. 315–322, 2002.

[83] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[84] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering*, pp. 867–872, 2008.

[85] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse, "An empirical comparison between direct and indirect test result checking approaches," in *Proceedings of the 3rd International Workshop on Software Quality Assurance*, pp. 6–13, ACM, 2006.

[86] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?," *IEEE Transactions on Software Engineering*, vol. 40, pp. 4–22, Jan 2014.

[87] K. S. Mishra and G. Kaiser, "Effectiveness of teaching metamorphic testing," Tech. Rep. CUCS-020-12, Department of Computer Science, Columbia University, 2012.

[88] C. Murphy, *Metamorphic Testing Techniques to Detect Defects in Applications without Test Oracles*. PhD dissertation, Columbia University, 2010.

[89] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

[90] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34 –41, Apr. 1978.

[91] U. Kanewala and J. Bieman, "Techniques for testing scientific programs without an oracle," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, pp. 48–57, May 2013.

[92] U. Kanewala, "Techniques for automatic detection of metamorphic relations," in *Proceedings of the Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 237–238, IEEE, 2014.

[93] R. A. Oliveira, U. Kanewala, and P. A. Nardi, "Automated test oracles: State of the art, taxonomies, and trends," *Adavances in computers*, vol. 95, pp. 113–199, 2014.

[94] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering*, pp. 85–103, IEEE Computer Society, 2007.

[95] K. M. Borgwardt, C. S. Ong, S. Schnauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21, no. suppl 1, pp. i47–i56, 2005.

APPENDIX A

Functions used in the empirical studies

Table A.1. Details of the functions used in the experiment in Chapter 3 (P: Permutative, A: Additive, I: Inclusive, ✓: positive example for the relation, ×: negative example, -: not used as an example)

No.	Function Name	P	A	I
1.	*add_values* (Add elements in an array)	✓	✓	✓
2.	*add_two_array_values* (Adds elements at given index in 2 arrays)	×	✓	✓
3.	*bubble_sort* (Implements bubble sort)	✓	✓	×
4.	*shell_sort* (Implements of shell sort)	✓	✓	×
5.	*binary_search*	×	✓	✓
6.	*sequential_search*	×	✓	✓
7.	*selection_sort* (Implements selection sort)	✓	✓	×
8.	*dot_product*	×	✓	✓
9.	*array_div* (Divide array elements by k)	×	✓	-
10.	*set_min_val* (Set array elements less than k equal to k)	×	×	✓
11.	*find_min* (Find minimum value in an array)	✓	✓	✓
12.	*find_diff* (Element-wise difference in two arrays)	×	✓	-
13.	*array_copy* (Deep copy an array)	×	-	✓
14.	*copy_array_part*	×	-	✓
15.	*find_euc_dist* (Euclidean distance between two vectors)	×	✓	✓
16.	*find_magnitude* (magnitude of a vector)	✓	✓	✓
17.	*manhattan_dist* (Manhattan distance between two vectors)	×	×	✓
18.	*average*	✓	✓	✓
19.	*dec_array* (Decrement elements by k)	×	✓	-
20.	*find_max* (Find the maximum value)	✓	✓	✓
21.	*find_max2* (maximum value of addition of two consecutive elements in an array)	×	✓	×
22.	*quick_sort* (Implements quick sort)	✓	✓	×
23.	*variance*	✓	✓	×
24.	*insertion_sort* (Implements insertion sort)	✓	✓	×
25.	*heap_sort* (Implements heap sort)	✓	✓	×
26.	*merge_sort* (Implements of merge sort)	✓	✓	×
27.	*geometric_mean*	✓	✓	×
28.	*mean_absolute_error*	×	×	✓
29.	*select_k* (Find the k^{th} largest value from a set of numbers)	✓	✓	×
30.	*find_median*	✓	✓	×
31.	*cartesian_product* (Cartesian product between two sets)	✓	×	✓
32.	*reverse* (Reverse an array)	×	-	-
33.	*check_equal_tolerance* (Checks element-wise equality within a given tolerance)	×	×	-
34.	*check_equal* (Element-wise equality between two sets of integers)	×	×	-
35.	*weighted_average*	×	✓	✓
36.	*count_k* (Occurrences of k in an array)	✓	×	✓
37.	*bitwise_and*	×	-	-
38.	*bitwise_or*	×	-	-

TABLE A.2. Details of the functions used in the experiment in Chapter 3 (P: Permutative, A: Additive, I: Inclusive, ✓: +ve example for the relation, ×: -ve example, -: not used as an example)

No.	Function Name	P	A	I
39.	*bitwise_xor*	×	-	-
40.	*bitwise_not*	×	-	-
41.	*clip* (Values outside a given interval clipped to the edges of the interval in an array)	×	×	-
42.	*elementwise_max* (Element-wise maximum)	×	×	-
43.	*elementwise_min* (Element-wise minimum)	×	×	-
44.	*cnt_nzeroes* (Number of non-zero elements in an array)	✓	×	✓
45.	*cnt_zeros* (Number of zero elements in a given array	✓	×	✓
46.	*elementwise_equal* (Check for element-wise equality and returns a boolean array)	×	×	-
47.	*elementwise_not_equal* (Check two for element-wise for nonequality)	×	×	-
48.	*hamming_dist*	-	×	-

TABLE A.3. Functions used in the experiment in Chapter 4.

Open source project	Functions used in the experiment
Colt Project	min, max, covariance, durbinWatson, lag1, meanDeviation, product, weightedMean, autoCorrelation, binarySearchFromTo, quantile, sumOfLogarithms, kurtosis, pooledMean, sampleKurtosis, sampleSkew, sampleVariance, pooledVariance, sampleWeightedVariance, skew, standardize, weightedRMS, harmonicMean, sumOfPowerOfDeviations, power, square, winsorizedMean, polevl
Apache Mahout	add, cosineDistance, manhattanDistance, chebyshevDistance, tanimotoDistance, hammingDistance, sum, dec, errorRate
Apache Commons Mathematics Library	errorRate, scale, eucleadianDistance, distance1, distanceInf, ebeAdd, ebeDivide, ebeMultiply, ebeSubtract, safeNorm, entropy, g, calculateAbsoluteDifferences, calculateDifferences, computeDividedDifference, computeCanberraDistance, evaluateHoners, evaluateInternal, evaluateNewton, mean, meanDifference, variance, varianceDifference, equals, checkNonNegative, checkPositive, chiSquare, evaluateWeightedProduct, partition, geometricMean, weightedMean, median, dotProduct
Functions from the previous study [69]	reverse, add_values, bubble_sort, shell_sort, sequential_search, selection_sort, array_calc1, set_min_val, get_array_value, find_diff, array_copy, find_magnitude, dec_array, find_max2, insertion_sort, mean_absolute_error, check_equal_tolerance, check_equal, count_k, clip, elementwise_max, elementwise_min, count_non_zeroes, cnt_zeroes, elementwise_equal, elementwise_not_equal, select

APPENDIX B

Definitions of graph kernels

B.1. Definition of the random walk kernel

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graph representations of programs as described in Section 3.2.1. Consider two walks, $walk_1$ in G_1 and $walk_2$ in G_2. $walk_1 = (v_1^1, v_1^2, ..., v_1^{n-1}, v_1^n)$ where $v_1^i \in V_1$ for $1 \leq i \leq n$. $walk_2 = (v_2^1, v_2^2, ..., v_2^{n-1}, v_2^n)$ where $v_2^i \in V_2$ for $1 \leq i \leq n$. $(v_1^i, v_1^{i+1}) \in E_1$ and $(v_2^i, v_2^{i+1}) \in E_2$. Then the kernel value of two graphs can be defined as

$$k_{rw}(G_1, G_2) = \sum_{walk_1 \in G_1} \sum_{walk_2 \in G_2} k_{walk}(walk_1, walk_2), \tag{7}$$

where the walk kernel k_{walk} can be defined as

$$k_{walk}(walk_1, walk_2) = \prod_{i=1}^{n-1} k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})). \tag{8}$$

The kernel for each step will be defined using the kernel values of the two node pairs and the edge pair of the considered step as follows:

$$k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = k_{node}(v_1^i, v_2^i) * k_{node}(v_1^{i+1}, v_2^{i+1}) * k_{edge}((v_1^i, v_1^{i+1}), (v_2 i, v_2^{i+1})). \tag{9}$$

We defined two node kernels: k_{node^1} and k_{node^2} to get the similarity score between two nodes. We use k_{node^1} and k_{node^2} in place of k_{node} in equation (9). k_{node^1} checks the similarity of the node labels.

$$k_{node^1}(v_i, v_j) = \begin{cases} 1 & \text{if } label(v_i) = label(v_j), \\ 0 & otherwise. \end{cases} \tag{10}$$

In the second node kernel, k_{node^2} we considered whether the operation performed in the two nodes are in the same group if the node labels are not equal. For this study we grouped the mathematical operations using commutative and associative properties.

$$k_{node^2}(v_i, v_j) = \begin{cases} 1 & \text{if } label(v_i) = label(v_j), \\ 0.5 & \text{if } group(v_i) = group(v_j) \text{ and } label(v_i) \neq label(v_j), \\ 0 & \text{if } group(v_i) \neq group(v_j) \text{ and } label(v_i) \neq label(v_j). \end{cases} \tag{11}$$

The edge kernel, k_{edge} is defined as follows:

$$k_{edge}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = \begin{cases} 1 & \text{if } label(v_1^i, v_1^{i+1}) = label(v_2^i, v_2^{i+1}), \\ 0 & otherwise. \end{cases} \tag{12}$$

We used the *direct product graph* approach presented by Gärtner et al. [71] with the modification introduced by Borgwardt et al. [95] for calculating all the walks within two graphs. The direct product graph of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is denoted by $G_1 \times G_2$. The nodes and edges of the direct product graph are defined as follows:

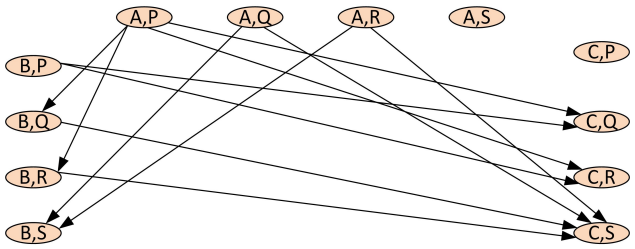


FIGURE B.1. Direct product graph of G_1 and G_1 in Table 4.3

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

$$E_X(G_1 \times G_2) = \{((v_1^1, v_1^2), (v_2^1, v_2^2)) \in V^2(G_1 \times G_2) : (v_1^1, v_1^2) \in E_1 \wedge (v_2^1, v_2^2) \in E_2 \wedge$$

$$label(v_1^1, v_1^2) = label(v_2^1, v_2^2)\}$$

Figure B.1 shows the direct product graph of the two graphs G_1 and G_2 in Table 4.3. As shown in figure B.1, the direct product graph has a node for each pair of nodes in G_1 and G_2. There is an edge between two nodes in the product graph if there are edges between the two corresponding pairs of the nodes in G_1 and G_2. Taking a walk in the direct product graph is equivalent to taking simultaneous walks in G_1 and G_2. Consider the walk $AP \rightarrow BQ \rightarrow CS$ in the direct product graph. This walk represents taking the walks $A \rightarrow B \rightarrow C$ in G_1 and $P \rightarrow Q \rightarrow S$ in G_2 simultaneously. Therefore by modifying the adjacency matrix of the direct product graph to contain similarity scores between steps, instead of having one/zero values, we can use the adjacency matrix of the direct product graph to efficiently compute the random walk kernel value of two graphs. We present the definition of the direct product graph and how it is used to compute the random walk kernel in Section B.1.

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

$$k_{rw}(G_1, G_2) = \sum_{i,j=1}^{V_X} \left[\sum_{n=0}^{\infty} \lambda^n A_X^n \right]_{ij} \tag{14}$$

where A_X denotes the adjacency matrix of the direct product graph and $1 > \lambda \geq 0$ is a weighting factor. To make the sum finite, we limited n in Equation (14) to 10 in our experiments. The adjacency matrix of the product graph is modified as follows to include k_{step} defined in Equation (9):

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

B.2. Definition of the graphlet kernel

Shervashidze et al. [68] developed a graph kernel that compares all subgraphs with $k \in \{3, 4, 5\}$ nodes. The authors refer to this kernel as the *graphlet kernel*, which was developed for unlabeled graphs. We extended the graphlet kernel for directed labeled graphs since the labels in our graph based program represent important semantic information about the function. We first describe the original graphlet kernel and then describe the modifications we made to it.

Let a graph be a pair $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ are the n vertices and $E \subseteq V \times V$ is the set of edges. Given $G = (V, E)$ and $H = (V_H, E_H)$, H is said to be a subgraph of G iff there is an injective mapping $\alpha : V_H \to V$ such that $(v, w) \in E_H$ iff $(\alpha(v), \alpha(w)) \in E$. If H is a subgraph 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 there exists a bijective mapping $g : V_1 \rightarrow V_2$ such that $(v_i, v_j) \in E_1$ iff $(g(v_i), g(v_j)) \in E_2$. If G_1 and G_2 are isomorphic, it is denoted by $G_1 \simeq G_2$ and g is called the isomorphism function.

Let $\mathcal{M}_1^k$ and $\mathcal{M}_2^k$ be the set of size k subgraphs of the graphs G_1 and G_2 respectively. Let $S_1 = (V_{S_1}, E_{S_1}) \in \mathcal{M}_1^k$ and $S_2 = (V_{S_2}, E_{S_2}) \in \mathcal{M}_2$. Then the graphlet kernel, $k_{graphlet}(G_1, G_2)$ is computed as

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in \mathcal{M}_1^k} \sum_{S_2 \in \mathcal{M}_2^k} \delta(S \simeq S_2) \tag{15}$$

where

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

The kernel in Equation (15) is developed for unlabeled graphs. To consider the node labels and edge labels we modified the kernel in 15 as follows:

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in \mathcal{M}_1^k} \sum_{S_2 \in \mathcal{M}_2^k} k_{subgraph}(S_1, S_2) \tag{16}$$

where

(17)

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

We used Equation (16) to compute the graphlet kernel value for a pair of programs represented in the graph based representation described in Section 3.2.1. Similar to the random walk kernel, k_{node} in Equation (17) is replaced by k_{node^1} and k_{node^2} defined in Equation (10) and Equation(11) respectively. Equation (12) defines k_{edge}.

B.3. Kernel normalization

We normalize each kernel such that each example has a unit norm by the expression [73, 67]:

$$k'_{graph}(G_1, G_2) = \frac{k_{graph}(G_1, G_2)}{\sqrt{k_{graph}(G_1, G_1) k_{graph}(G_2, G_2)}} \tag{18}$$