Preprint: Accepted for publication October 2015

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2015; **00**:1–30
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Predicting Metamorphic Relations for Testing Scientific Software: A Machine Learning Approach Using Graph Kernels

Upulee Kanewala*, James M. Bieman, and Asa Ben-Hur

Computer Science Department, Colorado State University, USA

SUMMARY

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 in some programs, and especially in scientific software. *Metamorphic testing* is a method for automating the testing process for programs without test oracles. This technique operates by checking whether the program behaves according to properties called *metamorphic relations*. A metamorphic relation describes the change in output when the input is changed in a prescribed way. Unfortunately, finding the metamorphic relations satisfied by a program or function remains a labor intensive task, which is generally performed by a domain expert or a programmer. In this work we propose a machine learning approach for predicting metamorphic relations that uses a graph-based representation of a program to represent control flow and data dependency information. In earlier work we found that simple features derived from such graphs provides good performance. An analysis of the features used in this earlier work led us to explore the effectiveness of several representations of those graphs using the machine learning framework of graph kernels, which provide various ways of measuring similarity between graphs. Our results show that a graph-kernel that evaluates the contribution of all paths in the graph has the best accuracy and that control flow information is more useful than data dependency information. The data used in this study is available for download at http://www.cs.colostate.edu/saxs/MRpred/functions.tar.gz to help researchers in further development of metamorphic relation prediction methods.
Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Metamorphic testing, Metamorphic relations, Graph kernels, Support vector machines

1. INTRODUCTION

Custom scientific software is widely used in science and engineering. Such software plays an important role in critical decision making in fields such as the nuclear industry, medicine and the military [1, 2]. In addition, results obtained from such software are used as evidence for research publications [2]. But due to the lack of systematic testing in scientific software, subtle faults can remain undetected. These subtle faults can cause incorrect program output without causing

*Correspondence to: upuleegk@cs.montana.edu

Copyright © 2015 John Wiley & Sons, Ltd.

a program to crash. For example, one-off errors caused loss of precision in seismic data processing programs [3], and software faults compromised the performance of coordinate measuring machines (CMMs) [4]. In addition, there have been multiple retractions of published work due to software faults [5]. Hatton and Roberts [6] found that several geoscience software systems designed for the same task produced reasonable yet essentially different results. There were situations where scientists believed that they needed to modify the physics model or develop new algorithms but later discovered that the real problem was small faults in the code [7]. Therefore it is important to conduct comprehensive automated testing on scientific software to ensure that they are 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" [8] where an oracle is unavailable or too difficult to implement. In such situations, a domain expert must manually check that the output produced from the application is correct for a selected set of inputs. Further, Sanders et al. [1] found that due to a lack of background knowledge in software engineering, scientists often 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 based on personal opinions and recommendations [9]. Therefore undetected subtle faults can affect findings of multiple studies that use the same scientific software. Techniques that can make it easier to test software without oracles are clearly needed [10].

Metamorphic testing is a technique introduced by Chen et al. [11], that can be used to test programs that do not have oracles. This technique operates by checking whether the program being tested behaves according to an expected set of properties known as *metamorphic relations*. A *metamorphic relation (MR)* specifies how a particular change to the input of the program should change the output [12]. 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 an MR for testing this program.

Violation of an MR occurs when the change in the output differs from what is specified by the considered MR. Satisfying a particular MR does not guarantee that the program is implemented correctly. However, a violation of an MR indicates that the program contains faults. Previous studies show that metamorphic testing can be an effective way to test programs without oracles [12, 13]. Enumerating a set of MRs that should be satisfied by a program is a critical initial task in applying metamorphic testing [14, 15]. Currently, a tester or a developer has to manually identify MRs using her knowledge of the program being tested; this manual process can easily miss some of the important MRs that could reveal faults.

In previous work we presented machine learning prediction models that can automatically predict MRs of previously unseen functions using a set of features extracted from their control flow graphs [16]. In this work we

Copyright © 2015 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

1. perform a feature analysis to identify the most predictive features from the features developed in our previous study. The results of the feature analysis showed that paths in control flow graphs are highly important for predicting MRs.
2. develop an efficient method for measuring similarity between programs represented as graphs. This method extends the explicitly extracted features by incorporating more information about the function under consideration such as mathematical properties of the operations within the functions. Our results show that these graph kernels lead to higher prediction accuracy.
3. evaluate the effect of using data dependency information beyond using control flow information used in our previous study. Our results show that data dependency information alone are not as predictive as control flow information for metamorphic relation prediction.

The remainder of this paper is organized as follows: Section 2 describes details about metamorphic testing. Our approach including the detailed information about how we apply graph kernels are described in Section 3. Section 4 and Section 5 describes our experimental setup and the results of our empirical studies, respectively. We discuss threats to validity in Section 6. Section 7 presents the related work. Section 8 provides our conclusions and future work.

2. METAMORPHIC TESTING

Metamorphic testing was introduced by Chen et al. [11] to alleviate the oracle problem. It supports the creation of follow-up test cases from existing test cases [11, 12] through the following process:

1. Identify an appropriate set of MRs that the program being tested should satisfy.
2. Create a set of *initial* test cases using techniques such as random testing, structural testing or fault based testing.
3. Create *follow-up* test cases by applying the input transformations required by the identified MRs in Step 1 to each initial test case.
4. Execute the corresponding initial and follow-up test case pairs to check whether the output change complies with the change predicted by the MR. A run-time violation of an MR during testing indicates a fault or faults in the program being tested.

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

Consider the function in Figure 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* MR in Table I. Further, adding a positive integer k to every element in a should increase the result by $k \times length(a)$. This is the *additive* MR in Table I. 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.

Identifying a set of metamorphic relations that should be satisfied by a program being tested is a critical initial step in metamorphic testing. Often these metamorphic relations are identified manually by the developers in an ad-hoc manner. Liu et al. introduced a systematic method for constructing new MRs based on the composition of already identified MRs [17]. In our

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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

Figure 1. Function for calculating the sum of elements in an array.

previous work, we introduced a machine learning based approach to automatically predict likely metamorphic relations for given a function. We extracted set of features from control flow graphs of functions and used them build machine learning classifiers. To our knowledge, that is the first attempt for automatically predicting likely MRs [16]. Zhang et al. developed a search based approach for the inference of polynomial MRs [18]. With polynomial MRs, inputs and outputs of initial and follow-up test cases need to be related by a polynomial equations. Our approach differs in several ways from the approach proposed by Zhang et al. First our approach does not enforce a restriction on the type of MRs that can be predicted. Also our approach only uses static properties of the program being tested, while Zang et al. use a dynamic approach where the program being tested is executed with test inputs in order to find MRs. Thus the quality of the MRs found depends on the test inputs used. Our approach does not require execution of the code. These two approaches can complement each other when they are applied to obtain MR predictions for a given program.

Metamorphic testing has been used to test applications without oracles in different areas such as machine learning applications [14], health care simulations [19], Monte Carlo modeling [20], bioinformatics programs [21], computer graphics [22], programs with partial differential equations [23], and variability analysis tools [24]. Recently, metamorphic testing has been used to validate compilers [25] and cloud models[26]. Metamorphic testing has been also used to conduct model-based testing on NASA's Data Access Toolkit (DAT) [27]. All of these applications can benefit from our method for automatically finding likely metamorphic relations.

3. APPROACH

In this section we present the details of the approach that we developed to automatically predict metamorphic relations for a given function. We first provide a motivating example that demonstrates the intuition behind this novel graph kernel based approach. Then we explain the overall view of the approach followed by a detailed description of the individual steps.

3.1. Motivating example

Figure 2 displays a function that finds the maximum value of an array. Figure 3 displays a function that calculate the average of an array. Figure 4 displays a function for calculating the cumulative difference between consecutive array elements. All these functions take a double array as the input

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

and produce a double value as the output. Figure 5 depicts the control flow graphs of the three functions without the error handling code. Consider the permutative metamorphic relation, which states that if the elements in the input are randomly permuted, the output should remain the same. The functions *max* and *average* satisfy the permutative metamorphic relation, but it is not satisfied by the *calcRun* function.

Considers the CFGs for the functions *max* and *average*. They both have very similar overall structures and they both satisfy the permutative metamorphic relation. So functions that have similar CFGs may have similar MRs. But the overall structure of the *calcRun* function's CFG is also similar to both these CFGs as well. For example, all of them have a single loop that traverses through the input array. But the *calcRun* function does not satisfy the permutative metamorphic relation. Therefore by only looking at the similarities in overall structure of the CFGs, we cannot effectively determine the metamorphic relations satisfied by them.

By observing the three CFGs in Figure 5 it can be observed that the satisfiability of the permutative metamorphic relation is determined by the differences in the operations performed inside the loop. While *max* performs a comparison over the array elements, *calcRun* performs a calculation between consecutive array elements. If we only consider high level properties of the function such as number of loops or types of operations we would not be able to capture this difference. Therefore it is important to include information about the sequence of operations performed in a control flow path in the features used to create machine learning models, and to include properties of individual operations in the function. Based on this intuition, we developed two types of features called node features and path features in our previous work [16]. In this work we propose a novel method based on graph kernels for implicitly extracting features from graph based representations. These graph kernels are natural extensions of the features used in our previous work. But when compared with the explicitly extracted node and path features, graph kernels support incorporating similarities between operations in the functions as explained in Section 3.4.1 and Section 3.4.2.

```
double max(double[] a) {
  int size = a.length;
  if (size==0) error;
  double max = a[size-1];
  for (int i = size-1; --i >= 0;) {
    if (a[i] > max) max = a[i];
  }
  return max;
}
```

Figure 2. Function for finding the maximum element in an array

```
double average(double[] a) {
  int size = a.length;
  if (size==0) error;
  double sum = 0;
  for (int i = 0; i < size;i++) {
    sum=sum+a[i];
  }
  double avg=sum/size;
  return avg;
}
```

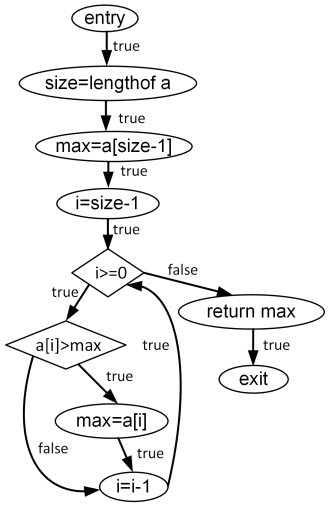
Figure 3. Function for finding the average of an array of numbers

Copyright © 2015 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

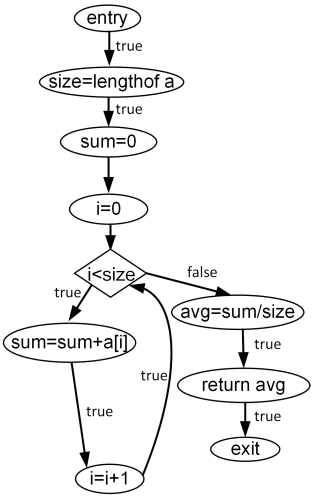
DOI: 10.1002/stvr

```
double calcRun(double[] a) {
  int size = a.length;
  if (size < 2) error;
  double run = 0;
  for(int i=1; i<size; ++i) {
    double x = a[i] - a[i-1];
    run += x*x;
  }
  return run;
}
```

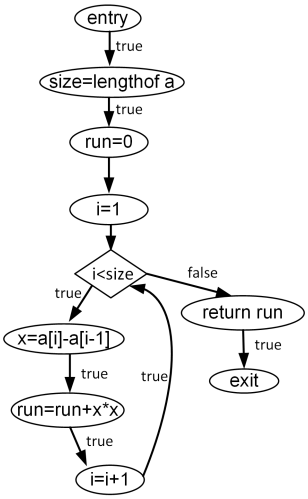
Figure 4. Function for calculating the running difference of the elements in an array



(a) CFG for the function in Figure 2



(b) CFG for the function in Figure 3



(c) CFG for the function in Figure 4

Figure 5. CFGs for the functions max, average, and calcRun

Copyright © 2015 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

3.2. Method overview

Our proposed approach uses machine learning methods to train binary classifiers to predict metamorphic relations. Figure 6 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 which provides a similarity score for each pair of functions in the training set. Then the computed graph kernel is used by a support vector machine (SVM) to create a predictive model. During the *testing phase*, we use the trained model to predict whether a previously unseen function satisfies the considered metamorphic relation. For the interested reader we provide background on kernel methods in Appendix A.

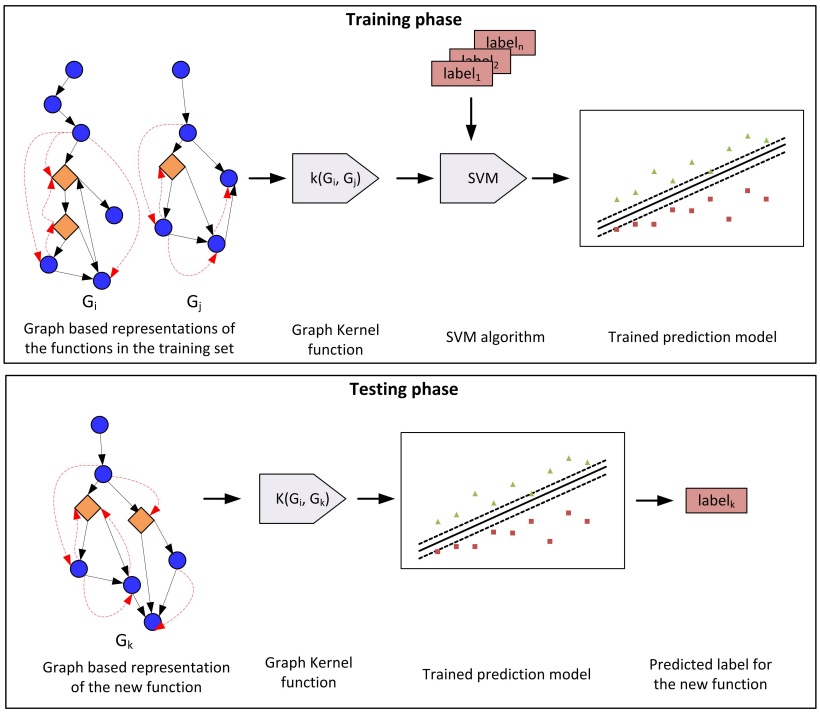


Figure 6. Overview of the approach.

3.3. Function Representation

We used the following graph-based representation of a function that contains both control flow information and data dependency information: the graph $G_f = (V, E)$ of a function f is a directed graph, where each node $v_x \in V$ 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

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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 [28].

We used the *Soot*[†] framework to create this graph based representation. *Soot* generates control flow graphs (CFG) in *Jimple* [29], a typed 3-address intermediate representation, where each CFG node represents an atomic operation. Consequently, each node in the graph is labeled with the atomic operation performed. 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 7 displays the graph based representation created for the function in Figure 1.

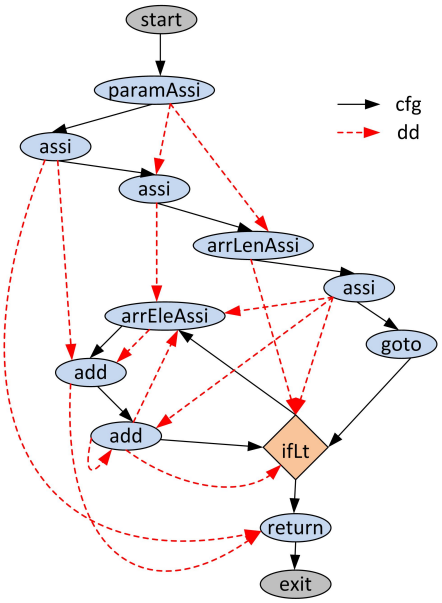


Figure 7. Graph representation of the function in Figure 1. cfg: control flow edges. dd: data dependency edges.

3.4. Graph Kernels

We define two graph kernels for the graph representations of the functions presented in Section 3.3: the *random walk kernel* and the *graphlet kernel*. Each kernel captures different graph substructures as described next.

3.4.1. The random walk kernel. Random walk graph kernels [30, 31] count the number of matching walks in two labeled graphs. This is a natural extension of the path features we used in our previous study. In what follows we explain the idea, while the details are provided in Appendix B. The value of the random walk kernel between two graphs is computed by summing up the contributions of all walks in the two graphs. Each pair of walks is 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 8. Computing this kernel requires specifying

[†]http://www.sable.mcgill.ca/soot/

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

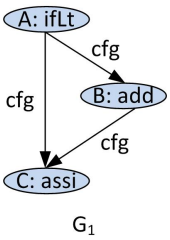
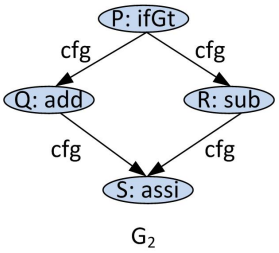
	
Walks of length 1: $A \to B$, $B \to C$, $A \to C$	Walks of length 1: $P \to Q$, $P \to R$, $Q \to S$, $R \to S$
Walks of length 2: $A \to B \to C$	Walks of length 2: $P \to Q \to S$, $P \to R \to S$
Computation of similarity score between two graphs (restricted to walks up to length 2): Computation 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 8. 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.

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 equation (5)). 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.

3.4.2. The graphlet kernel Subgraphs can directly capture important structures such as *if conditions*, that can be difficult to capture using random walks. Therefore we apply a kernel based on subgraphs that can be viewed as a natural extension of the node features used in our previous study. Results of the feature analysis presented in Section 5.1 showed that the majority of the top 1% of the features were node features. Therefore it is important to investigate whether extending these features would improve classifier accuracy.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The graphlet kernel computes a similarity score of a pair of graphs by comparing all subgraphs of limited size in the two graphs [32]. 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 9. 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 of 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 3.4.1. In Figure 9 we illustrate the computation of the kernel and the complete definition of the graphlet kernel is presented in Appendix C.

4. EXPERIMENTAL SETUP

This section describes the details of the code corpus and metamorphic relations used in our study. We also present details of the evaluation measures used in this study.

4.1. Research questions

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

- **RQ1:** What types of features are going to be most effective for predicting metamorphic relations?
- **RQ2.1:** What is the effect of using graph kernels to develop machine learning classifiers instead of explicitly extracted features?
  RQ2.2: Which substructures in graphs are most suitable for predicting metamorphic relations?
- **RQ3:** What static characteristics of programs are most effective in predicting metamorphic relations?

4.2. The code corpus

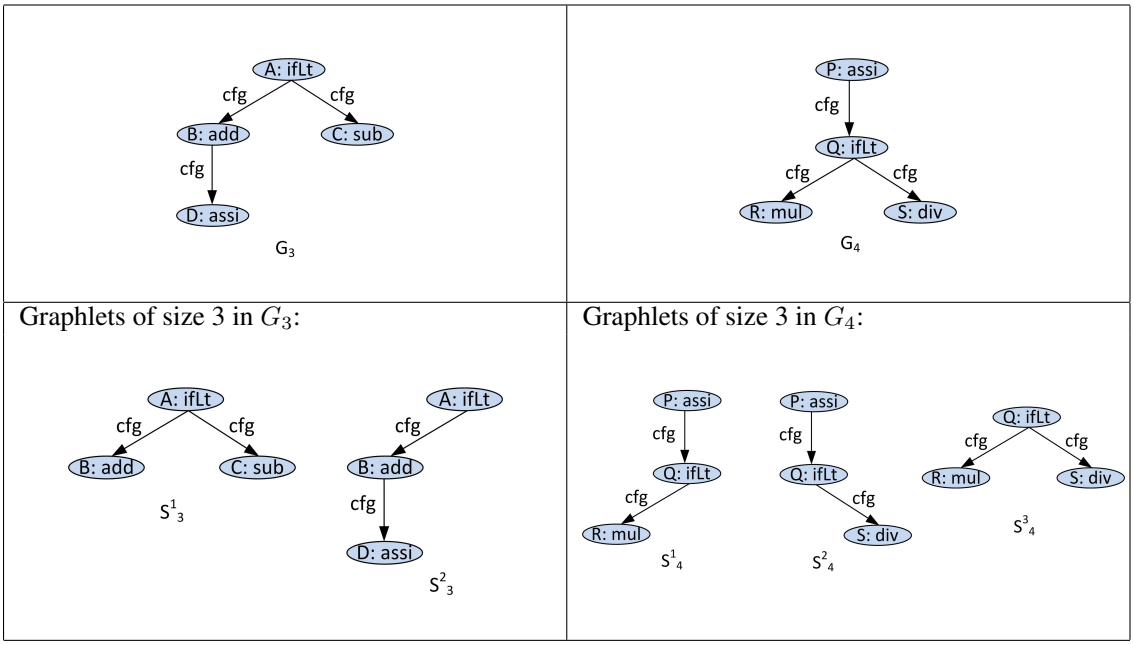
To measure the effectiveness of our proposed methods, we built a code corpus containing 100 functions that take numerical inputs and produce numerical outputs. We extended the corpus used in our previous study [16] with functions from the following open source projects:

1. The Colt Project[‡]: set of open source libraries written for high performance scientific and technical computing in Java.
2. Apache Mahout[§]: a machine learning library written in Java.
3. Apache Commons Mathematics Library[¶]: a library of mathematics and statistics components written in the Java.

[‡]http://acs.lbl.gov/software/colt/
[§]https://mahout.apache.org/
[¶]http://commons.apache.org/proper/commons-math/

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



Computation of similarity score between two graphs using graphlets:

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

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 9. Graphlet kernel computation for two 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.

We list these functions in Table VI in Appendix E. These functions and their graph representations can be accessed via the following URL: http://www.cs.colostate.edu/saxs/MRpred/functions.tar.gz. Functions in the code corpus perform various calculations using sets of numbers such as calculating statistics (e.g., average, standard deviation, kurtosis), calculating distances (e.g., Manhattan and Chebyshev), and searching/sorting, etc. Lines

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table I. 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 II. 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

of code (LOC) of these functions varied between 5 and 54 and the cyclomatic complexity varied between 1 and 11. The number of input parameters to each function varied between 1 and 5.

4.3. Metamorphic relations

We used the six metamorphic relations shown in Table I. These metamorphic relations were identified by Murphy et al. [33] for testing machine learning applications and are also commonly found in mathematical functions. We list the input modifications and the expected output modification of these metamorphic relations in Table I. For example, consider the inclusive metamorphic relation in Table I. The function in Figure 1 satisfies this metamorphic relation since adding a new element to the input array a should increase the output. Similarly, removing an element from the input array a should decrease the output. Therefore that function also satisfies the exclusive metamorphic relation. Metamorphic relations and the expected output changes presented in Table I will not be valid for any numerical functions in general. For example, the additive metamorphic relation will not be valid for non-monotonic functions. The output can either increase or decrease when a positive constant is added to the input of such a function.

A function f is said to satisfy (or exhibit) a metamorphic relation m in Table I, 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 [34, 35, 36]. 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 II reports the number of positive and negative instances for each metamorphic relation.

4.4. Evaluation procedure

We used 10-fold stratified cross validation in our experiments. In 10-fold cross validation the data set is randomly partitioned into 10 subsets. Then nine subsets are used to build the predictive model

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

(training) and the remaining subset is used to evaluate the performance of the predictive model (testing). This process is repeated 10 times in which each of the 10 subsets is used to evaluate the performance. In stratified 10-fold cross validation, the 10 folds are partitioned in such a way that the folds contain approximately the same proportion of positive instances (functions that exhibit a specific metamorphic relation) and negative instances (functions that do not exhibit a specific metamorphic relation) as in the original data set. Results are generated by averaging over 10 runs of cross validation.

We used nested cross validation to select the regularization parameter of the 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[‖] in this work.

We used two evaluation measures when reporting our results. The first evaluation measure is the *balanced success rate (BSR)* [37]. 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:

$$BSR = \tfrac{1}{2}(P(success|+) + P(success|-)),$$

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). AUC measures the probability that a randomly chosen negative example will have a lower prediction score than a randomly chosen positive example [38]. Therefore a higher AUC value indicates that the model has a higher predictive ability. A classifier with AUC = 1 is considered a perfect classifier, while a classifier that classifies randomly will have AUC = 0.5. We choose to use AUC as our primary evaluation metric since it does not depend on the discrimination threshold of the classifier and has been shown to be a better measure for comparing learning algorithms [38].

5. RESULTS

We present the results of our empirical studies in this section. We first present the results of the feature analysis that we conducted to identify the most effective features for predicting metamorphic relations. Then, we evaluate the effectiveness of the two graph kernels which are extensions of the node and path features, and compare their effectiveness with the features used in our previous work. Finally, we compare the effectiveness of control flow and data dependency information for predicting MRs.

5.1. Feature analysis

As a motivation for the use of graph kernels we conducted a feature analysis on the node/path features used in our previous study [16] to identify the most effective features for predicting metamorphic relations. For this analysis we extracted node/path features using only the control

[‖] http://pyml.sourceforge.net/

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

flow features in the graphs. We trained SVM classifiers using the extracted features for the six metamorphic relations used in this study. Then we sorted the features according to the weights assigned by the SVM classifier for each metamorphic relation and used the top weighted features in the analysis. We used the top 1% ($\simeq$ 30) out of a total of 3328 features.

We first looked at the median ranking of the node features and path features within the top 30 features for each metamorphic relation. Table III shows the median rankings of the node features and path features of the top 30 features. For all the metamorphic relations, path features were ranked higher than the node features. This shows that path features are very important for predicting metamorphic relations.

Table III. Rankings of node and path features in the top 30 weighted features

Metamorphic relation	Median ranking of path features	Median ranking of node features
Permutative	8.0	21.0
Additive	13.0	15.5
Multiplicative	11.5	21.5
Invertive	7.5	19.5
Inclusive	8.0	21.0
Exclusive	7.5	18.5

Table IV shows the median lengths of the paths that were in the top 30 features. For the six metamorphic relations used in the experiment, path lengths in the top 30 features varied between 5 and 7. Table V shows the percentage of path features that represent loops and the percentage of paths that were not related to loops within the top 30 features. For multiplicative and inclusive metamorphic relations the majority of the top path features were paths related to loops. For the other four metamorphic relations top path features were not related to loops. But for all the metamorphic relations, path features related to loops reported a higher median ranking than path features not related to loops. This indicates that paths that represents loops (or parts of loops) in a function are important for predicting metamorphic relations. This result is intuitive since in the majority of the functions in our code corpus, the calculations are implemented using loops. Therefore paths inside loops represent the characteristics of the calculations better and should be important for predicting metamorphic relations. This result also agrees with the intuition that we presented in Section 3.1.

Table IV. Median path lengths of top 30 features

Metamorphic relation	Median path length
Permutative	5
Additive	6.5
Multiplicative	7
Invertive	7
Inclusive	7
Exclusive	6.5

The results of this analysis show that paths in control flow graphs are highly important for predicting metamorphic relations. Further, the results show that paths that represent loops are even more important for making predictions than other paths in the control flow graph. This provides the motivation to investigate the effectiveness of the random walk kernel which is an extension to

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table V. Percentage and median ranking of loop/non-loop features in top 30 features

Metamorphic relation	Loop		Non-loop	
	Percentage	Median ranking	Percentage	Median ranking
Permutative	30.77	4	69.23	12
Additive	33.33	6	66.67	18
Multiplicative	54.17	9	45.83	20
Invertive	42.86	5.5	57.14	12
Inclusive	61.54	5	38.46	13
Exclusive	41.67	7	58.33	10

the explicitly extracted path features. We also investigated the effectiveness of the graphlet kernel which is a natural extension of the node features.

5.2. Graph kernel results

We compared the performance of the node/path features that we used in our initial study [16] 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 [16]. 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 [16].
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 10a and Figure 10b show the average BSR and average AUC for the three feature extraction approaches: node and path features, the random walk kernel, and the graphlet kernel. Figure 10b demonstrates that among the three approaches, the random walk kernel gives the best prediction accuracy for all the MRs when considering the AUC. For the BSR, this holds for five out of six MRs (see Figure 10a), probably because the classification threshold was not optimally chosen.

The improvement in performance over node/path features can be attributed to two factors: (1) The path features used in our previous work required exact matches between paths. But using a node kernel in the random walk kernel, we were able to allow "soft" matches between paths. 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 15 shows that using such high level properties of operations improves the accuracy of the prediction models. (2) The random walk kernel compares all the paths in a graph. The path features only used the shortest path between a pair of nodes.

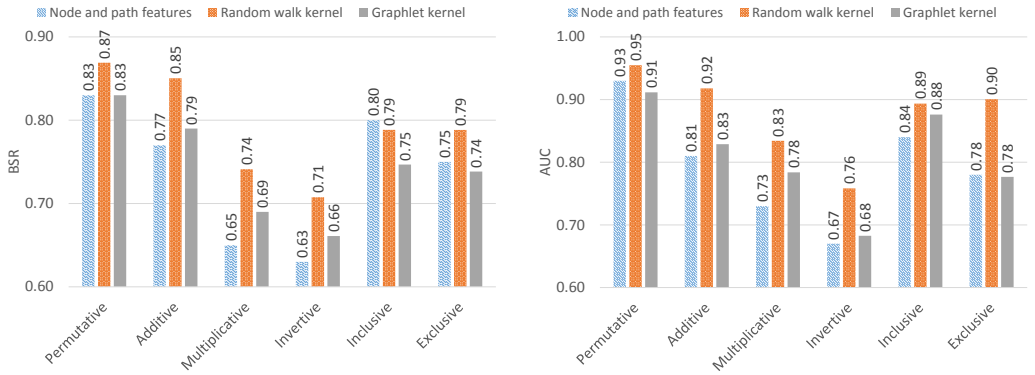
Our results show that the random walk kernel performs better than the graphlet kernel. This is consistent with the feature analysis results that we presented in Section 5.1, where the path features were ranked higher than the node features. The random walk kernel compares two control flow graphs based on their walks. A walk in the control flow graph corresponds to a potential execution

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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 compares 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. Our experiments showed that this combined kernel did not outperform the random walk kernel.



(a) BSR (b) AUC

Figure 10. Prediction accuracy of node and path features, the random walk kernel, and the graphlet kernel.

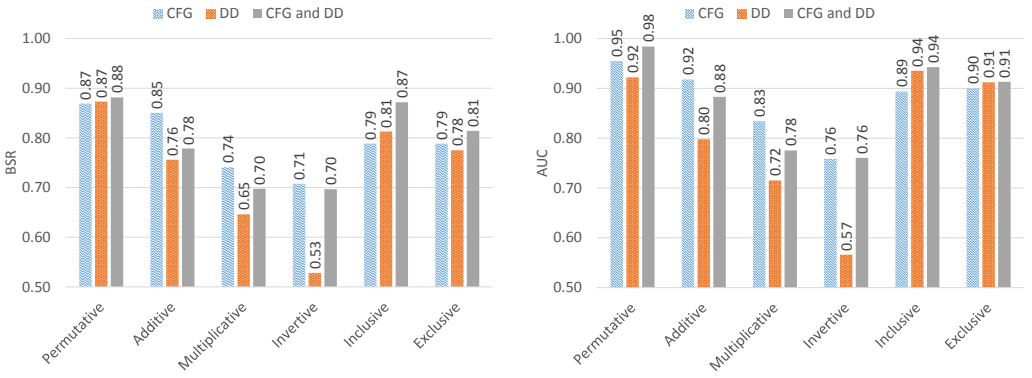
5.3. *Effectiveness of control flow and data dependency information*

Finally, we compared the effectiveness of the control flow and data dependency information of a function for predicting metamorphic relations. Figure 11 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 can be explained by the fact that the control flow graph directly represents information about the execution of the program compared to the data dependency information. 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.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



(a) BSR (b) AUC

Figure 11. Performance of control flow and data dependency information using the 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.

6. THREATS TO VALIDITY

External validity: We used a code corpus consisting of 100 mathematical functions for our empirical study. These functions differ in size and complexity, perform different functionalities, and were obtained from different open source libraries. Using a code corpus consisting of a larger number of functions will provide better results and minimize the threats to external validity.

In this work we presented a technique for predicting metamorphic relations at the function level. Therefore these predicted metamorphic relations techniques can be used as automated test oracles for conducting automated unit testing on scientific programs. 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 a higher number of metamorphic relations can be usually derived at the function level compared to deriving metamorphic relations for a whole program [39]. 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 [39]. Therefore the techniques that we present here should be effective in identifying faults when conducting unit testing.

Internal validity: 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 [40] 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.

Construct validity: Third party tools that we used in this work can pose threats to construct validity. We used the Soot framework to create the CFGs for the functions and annotate them with

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

data dependency information. Further we used the NetworkX[**] package for graph manipulation. To minimize these threats we verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool.

Conclusion validity: We used AUC value for evaluating the performance of the classifiers. We considered $AUC \geqslant 0.80$ as a good classifier. This is consistent with most of the machine learning literature.

7. RELATED WORK

Metamorphic testing has been used to test applications without oracles in different areas. Xie et al. [14] 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 [20]. Metamorphic testing has been used effectively in bioinformatics [21], computer graphics [22] and for testing programs with partial differential equations [23]. Murphy et al. [41] 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 [42]. 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 previous work showed that machine learning methods can be used to predict metamorphic relations in previously unseen functions [16]. We used features obtained from control flow graphs to train our prediction models. We showed that when predicting metamorphic relations, support vector machines achieved cross validation accuracies ranging from 83%-89% depending on the metamorphic relation.

Machine learning techniques have been used in different areas of software testing. For example, data collected during software testing involving test case coverage and execution traces can potentially be used in fault localization, test oracle automation, etc. [43]. Bowring et al. used program execution data to classify program behavior [44]. Briand et al. [45] used the C4.5 algorithm for test suite refinement. Briand et al. [46] used machine learning for fault localization to reduce the problems faced by other fault localization methods when several faults are present in the code. They used the C4.5 machine learning algorithm to identify failure conditions, and determine if a failure occurred due to the same fault(s).

Frounchi et al. [47] used machine learning to develop a test oracle for testing an implementation of an image segmentation algorithm. They used the C4.5 algorithm to build a classifier to determine whether a given pair of image segmentations are consistent or not. Once the classification accuracy is satisfactory, the classifier can check the correctness of the image segmentation program. Lo [48] used a SVM model for predicting software reliability. Wang et al. [49] used SVMs for creating

[**]https://networkx.github.io/

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

test oracles for reactive systems. They first collect test traces from the program being tested and obtain pass/fail decisions from an alternate method such as a domain expert. Then they train a SVM classification model using these labeled test traces. The trained SVM classifier works as the oracle for classifying previously unseen test traces. We did not find any applictions of graph kernels in software engineering.

8. CONCLUSIONS AND FUTURE WORK

Metamorphic testing is a useful technique for testing programs for which an oracle is not available. Identifying a set of metamorphic relations that should be satisfied by the program is an important initial task that determines the effectiveness of metamorphic testing, which is currently done manually. Our work investigates novel machine learning based approaches using graph kernels for predicting metamorphic relations for a given function.

Our previous work showed that classification models created using a set of features extracted from control flow graphs are effective in predicting metamorphic relations. Further, we showed that these prediction models make consistent predictions even when the functions contain faults. We also showed that the predicted metamorphic relations can effectively identify faults.

In this work we conducted a feature analysis to identify the most effective features for predicting metamorphic relations. Results of the feature analysis showed that features created using paths in control flow graphs are the most important. To take better advantage of such features we developed two graph kernels that extend the explicitly extracted features and used them for predicting metamorphic relations. Results using a set of functions obtained from open source code libraries show that graph kernels improve the prediction accuracy of metamorphic relations when compared with explicitly extracted features. Among the two graph kernels the random walk kernel, which implicitly considers all walks in the graph that represents a function, was the most effective.

In addition, we compared the effectiveness of control flow and data dependency information for predicting metamorphic relations. Results of our empirical studies show that control flow information of a function is more effective than data dependency information for predicting metamorphic relations; in a few cases using both types of information improved accuracy.

This work can be extended in several directions. Our approach considers metamorphic relations separately; however, there is likely information that can be mined from the dependencies and correlations across different metamorphic relations. This can be accomplished using a joint modeling of these separate tasks using multi-task learning [50] or multi-label algorithms [51]. In addition incorporating dynamic properties of programs such as dynamic execution traces is likely to provide further improvement in predicting metamorphic relations. Finally, additional evaluation is required to see how this approach extends to programs that handle other types of inputs/outputs such as graphs and matrices.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

A. KERNEL METHODS

In what follows we provide relevant background on kernel methods. Kernel methods are an effective approach when the data do not have a natural representation in a fixed dimensional space such as programs represented as graphs.

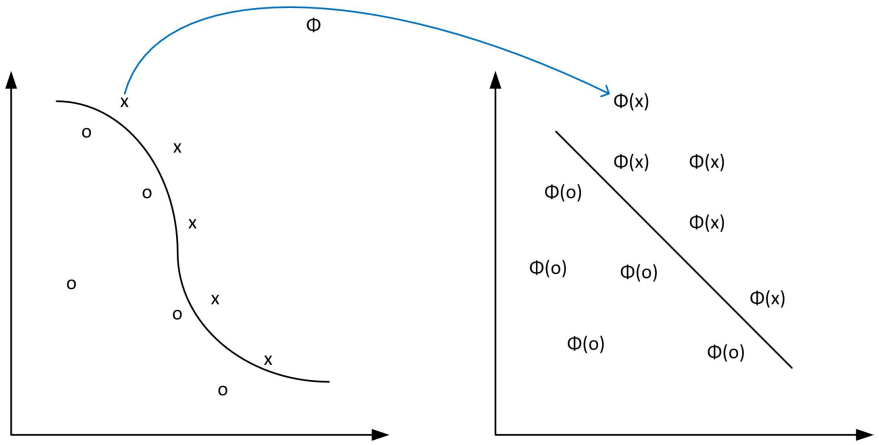


Figure 12. Function ϕ maps data into a new feature space such that a linear separation can be found.

Kernel methods perform pattern analysis by combining two ideas: (1) Embed the data in an appropriate feature space. (2) Use machine learning algorithms to discover linear relations in the embedded data [52]. Figure 12 depicts how the data is mapped to a feature space such 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* 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 [52]. 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 [32]: (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 paper we use the kernel based approach, specifically graph kernels and compare their performance

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

to the set of features we used in our previous work [16]. Graph kernels compute a similarity score for a pair of graphs by comparing their substructures, such as shortest paths [53], random walks [31], and subtrees [54]. 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).

B. 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 3.3. 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

$$k_{rw}(G_1, G_2) = \sum_{walk_1 \in G_1} \sum_{walk_2 \in G_2} k_{walk}(walk_1, walk_2), \tag{1}$$

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

$$k_{walk}(walk_1, walk_2) = \prod_{i=1}^{n-1} k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})). \tag{2}$$

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:

$$k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = k_{node}(v_1^i, v_2^i) * k_{node}(v_1^{i+1}, v_2^{i+1}) * k_{edge}((v_1^i, v_1^{i+1}), (v_2 i, v_2^{i+1})). \tag{3}$$

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 (3). k_{node^1} checks the similarity of the node labels.

$$k_{node^1}(v_i, v_j) = \begin{cases} 1 & \text{if } label(v_i) = label(v_j), \\ 0 & otherwise. \end{cases} \tag{4}$$

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.

$$k_{node^2}(v_i, v_j) = \begin{cases} 1 & \text{if } label(v_i) = label(v_j), \\ 0.5 & \text{if } group(v_i) = group(v_j) \text{ and } label(v_i) \neq label(v_j), \\ 0 & \text{if } group(v_i) \neq group(v_j) \text{ and } label(v_i) \neq label(v_j). \end{cases} \tag{5}$$

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

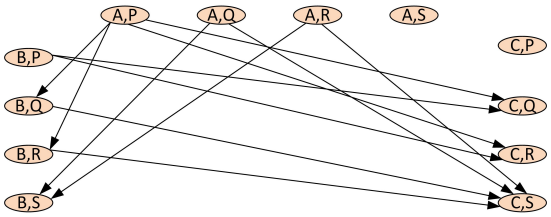


Figure 13. Direct product graph of G_1 and G_1 in Table 8

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

$$k_{edge}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = \begin{cases} 1 & \text{if } label(v_1^i, v_1^{i+1}) = label(v_2^i, v_2^{i+1}), \\ 0 & otherwise. \end{cases} \tag{6}$$

We used the *direct product graph* approach presented by Gärtner et al. [31] with the modification introduced by Borgwardt et al. [55] 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 next:

$$\begin{aligned} V_X(G_1 \times G_2) &= \{(v_1, v_2) \in V_1 \times V_2\} \\ E_X(G_1 \times G_2) &= \{((v_1^1, v_1^2), (v_2^1, v_2^2)) \in V^2(G_1 \times G_2) : (v_1^1, v_1^2) \in E_1 \wedge (v_2^1, v_2^2) \in E_2 \wedge \\ &\qquad label(v_1^1, v_1^2) = label(v_2^1, v_2^2)\} \end{aligned}$$

Figure 13 shows the direct product graph of the two graphs G_1 and G_2 in Table 8. As shown in figure 13, 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.

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

$$k_{rw}(G_1, G_2) = \sum_{i,j=1}^{V_X} \left[\sum_{n=0}^{\infty} \lambda^n A_X^n \right]_{ij}, \tag{8}$$

where A_X denotes the adjacency matrix of the direct product graph and $1 > \lambda \geq 0$ is a weighting factor. In our experiments we limited limited n in Equation (8) to 10. The adjacency matrix of the product graph is modified as follows to include k_{step} defined in Equation (3):

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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

C. DEFINITION OF THE GRAPHLET KERNEL

Shervashidze et al. [32] 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 $g : V_1 \to V_2$ 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

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in \mathcal{M}_1^k} \sum_{S_2 \in \mathcal{M}_2^k} \delta(S \simeq S_2). \tag{9}$$

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 (9) is developed for unlabeled graphs. To consider the node labels and edge labels we modified the kernel in 9 as follows:

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in \mathcal{M}_1^k} \sum_{S_2 \in \mathcal{M}_2^k} k_{subgraph}(S_1, S_2), \tag{10}$$

where

$$k_{subgraph}(S_1, S_2) = \begin{cases} \prod_{v \in V_{S_1}} k_{node}(v, g(v)) * \prod_{(v_i,v_j) \in E_{S_1}} k_{edge}((v_i, v_j), (g(v_i), g(v_j))) & \text{if } S_1 \simeq S_2, \\ 0 & otherwise. \end{cases} \tag{11}$$

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VI. A list of the functions used in the computational experiments.

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 [16]	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

We used Equation (10) to compute the graphlet kernel value for a pair of programs represented in the graph based representation described in Section 3.3. Similar to the random walk kernel, k_{node} in Equation (11) is replaced by k_{node^1} and k_{node^2} defined in Equation (4) and Equation(5) respectively. Equation (6) defines k_{edge}.

D. KERNEL NORMALIZATION

We normalize each kernel such that each example has a unit norm by the exprssion [37, 52]:

$$k'_{graph}(G_1, G_2) = \frac{k_{graph}(G_1, G_2)}{\sqrt{k_{graph}(G_1, G_1) k_{graph}(G_2, G_2)}}. \quad (12)$$

E. LIST OF FUNCTIONS

Table VI lists the functions used in our experiments.

F. EFFECTIVENESS OF THE RANDOM WALK KERNEL

For the evaluation of the random walk kernel, we first considered how prediction accuracy varies with the parameter λ of the kernel. For this evaluation we computed the random walk kernel

Copyright © 2015 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

values using only the control flow edges in the graphs. 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 D to normalize the random walk kernel. We selected the regularization parameter of the SVM using nested cross validation. Figure 14 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 (8) in Appendix B). Therefore, for predicting additive and multiplicative MRs, the contribution from long walks in the CFGs seems to 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 4 and equation 5 in Appendix B. We used nested cross validation to select the regularization parameter of the SVM and the λ parameter of the random walk kernel. Figure 15 shows the performance comparison between k_{node^1} and k_{node^2}. Figure 15b 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}.

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 16 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 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.

G. EFFECTIVENESS OF THE GRAPHLET KERNEL

For the evaluation of the graphlet kernel, we first considered how prediction accuracy varies with graphlet size. For this evaluation, we used only the control flow edges when computing the kernel. We used cosine normalization described in Appendix D 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 17a and Figure 17b shows how the average BSR and average AUC varies with graphlet size for each MR. When predicting the multiplicative MR, 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.

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}.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

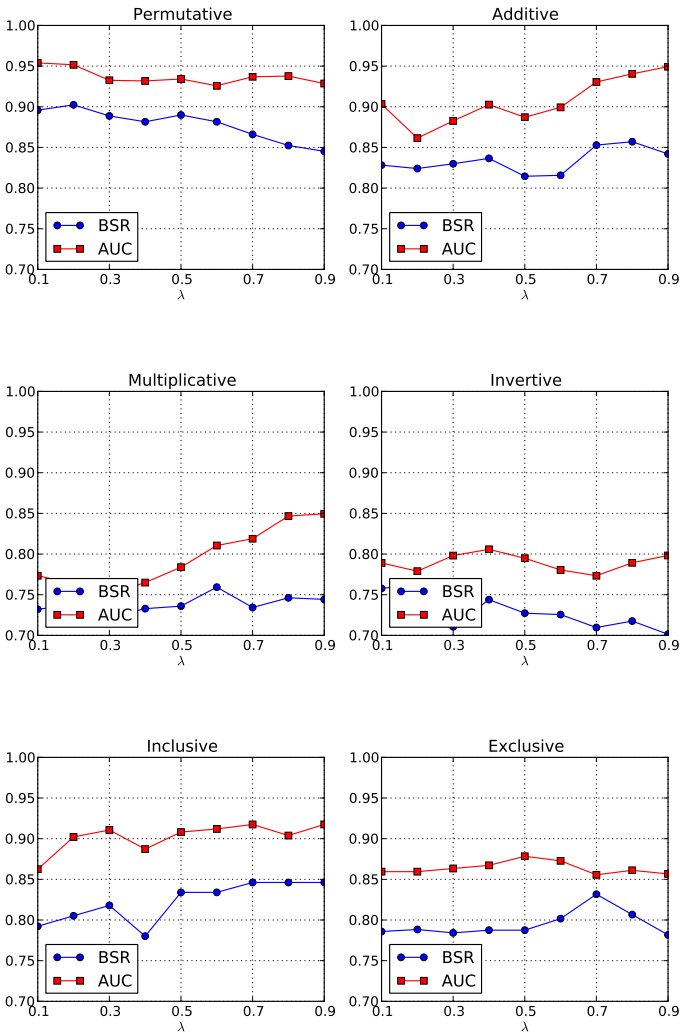


Figure 14. Variation of the BSR and the AUC with the parameter λ in the random walk graph kernel for each MR.

ACKNOWLEDGEMENT

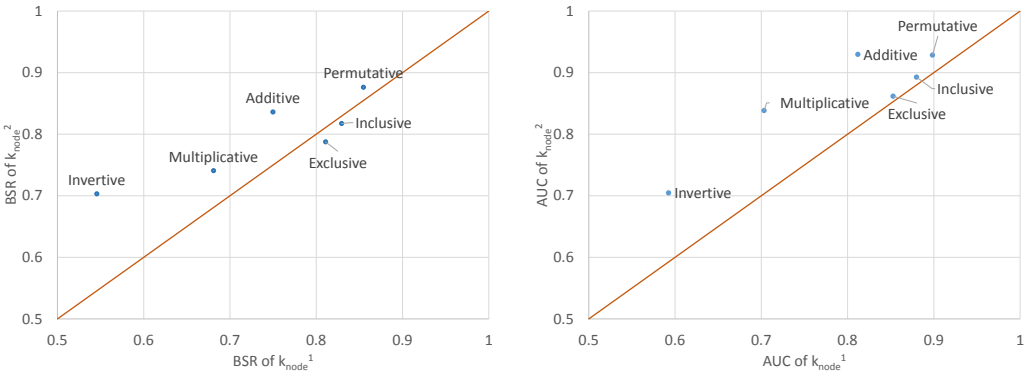
This project is supported by Award Number 1R01GM096192 from the National Institute Of General Medical Sciences. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute Of General Medical Sciences or the National Institutes of Health.

REFERENCES

1. Sanders R, Kelly D. The challenge of testing scientific software. *In Proc. of the Conference for the Association for Software Testing (CAST)*, Toronto, 2008; 30–36.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



(a) Comparison of BSR

(b) Comparison of AUC

Figure 15. Performance comparison of k_{node^1} and k_{node^2} for the random walk kernel.

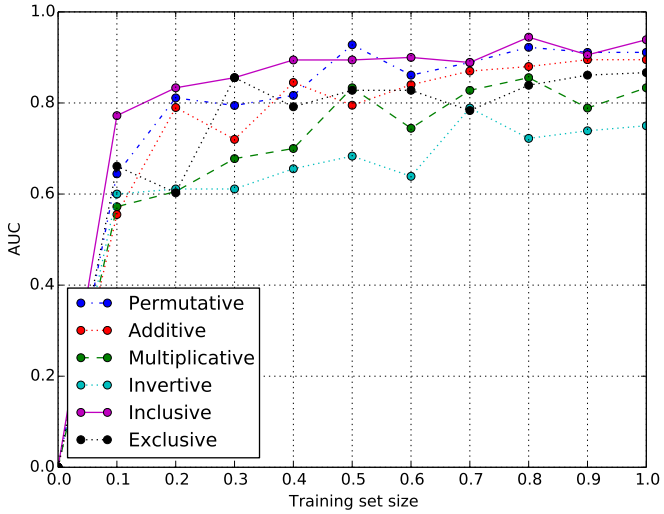
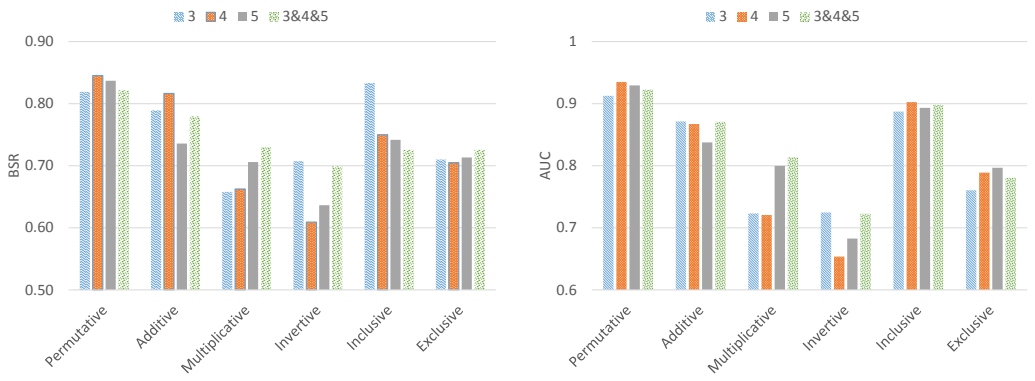


Figure 16. Learning curves for the six metamorphic relations; the x-axis is the fraction of the training set used for creating the model; the y-axis provides the accuracy measured by the classifer's AUC.

2. Sanders R, Kelly D. Dealing with risk in scientific software development. *IEEE Software* Jul– Aug 2008; **25**(4):21 –28, doi:10.1109/MS.2008.84.
3. Hatton L. The T experiments: errors in scientific software. *IEEE Computational Science Engineering* Apr– Jun 1997; **4**(2):27 –38, doi:10.1109/99.609829.
4. Abackerli AJ, Pereira PH, Calônego Jr N. A case study on testing CMM uncertainty simulation software (VCMM). *Journal of the Brazilian Society of Mechanical Sciences and Engineering* Mar 2010; **32**:8 – 14.
5. Miller G. A scientist's nightmare: Software problem leads to five retractions. *Science* 2006; **314**(5807):1856–1857, doi:10.1126/science.314.5807.1856. URL http://www.sciencemag.org/content/314/5807/1856.short.
6. Hatton L, Roberts A. How accurate is scientific software? *IEEE Transactions on Software Engineering* Oct 1994; **20**(10):785 –797, doi:10.1109/32.328993.
7. Dubois P. Testing scientific programs. *Computing in Science Engineering* Jul–Aug 2012; **14**(4):69 –73, doi: 10.1109/MCSE.2012.84.
8. Weyuker EJ. On testing non-testable programs. *The Computer Journal* 1982; **25**(4):465–470, doi:10.1093/comjnl/25.4.465. URL http://comjnl.oxfordjournals.org/content/25/4/465.abstract.

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



(a) BSR (b) AUC

Figure 17. Variation of performance with graphlet size. The bars are labeled with graphlet size (3, 4, or 5); the bar labeled 3&4&5 provides the accuracy of a model trained using all the graphlets.

9. Joppa LN, McInerny G, Harper R, Salido L, Takeda K, O'Hara K, Gavaghan D, Emmott S. Troubling trends in scientific software use. *Science* 2013; **340**(6134):814–815, doi:10.1126/science.1231535. URL http://www.sciencemag.org/content/340/6134/814.short.
10. Kanewala U, Bieman JM. Testing scientific software: A systematic literature review. *Information and Software Technology* 2014; **56**(10):1219 – 1232, doi:http://dx.doi.org/10.1016/j.infsof.2014.05.006. URL http://www.sciencedirect.com/science/article/pii/S0950584914001232.
11. Chen TY, Cheung SC, Yiu SM. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01*, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong 1998.
12. Chen TY, Tse TH, Zhou ZQ. Fault-based testing without the need of oracles. *Information and Software Technology* 2003; **45**(1):1–9.
13. Zhou ZQ, Huang DH, Tse TH, Yang Z, Huang H, Chen TY. Metamorphic testing and its applications. *In Proc. 8th International Symposium on Future Software Technology (ISFST 2004)*, Software Engineers Association, 2004.
14. Xie X, Ho JW, Murphy C, Kaiser G, Xu B, Chen TY. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 2011; **84**(4):544 – 558, doi:10.1016/j.jss.2010.11.920.
15. Chen TY, Huang DH, Tse TH, Zhou ZQ. Case studies on the selection of useful relations in metamorphic testing. *In Proc. 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, 2004; 569–583.
16. Kanewala U, Bieman J. Using machine learning techniques to detect metamorphic relations for programs without test oracles. *In Proc. 24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013; 1–10, doi:10.1109/ISSRE.2013.6698899.
17. Liu H, Liu X, Chen TY. A new method for constructing metamorphic relations. *In Proc. 12th International Conference on Quality Software (QSIC)*, 2012; 59–68, doi:10.1109/QSIC.2012.10.
18. Zhang J, Chen J, Hao D, Xiong Y, Xie B, Zhang L, Mei H. Search-based inference of polynomial metamorphic relations. *In Proc. 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, ACM: New York, NY, USA, 2014; 701–712, doi:10.1145/2642937.2642994. URL http://doi.acm.org/10.1145/2642937.2642994.
19. Murphy C, Raunak MS, King A, Chen S, Imbriano C, Kaiser G, Lee I, Sokolsky O, Clarke L, Osterweil L. On effective testing of health care simulation software. *In Proc. 3rd Workshop on Software Engineering in Health Care*, SEHC '11, ACM: New York, NY, USA, 2011; 40–47, doi:10.1145/1987993.1988003.
20. Ding J, Wu T, Wu D, Lu JQ, Hu XH. Metamorphic testing of a Monte Carlo modeling program. *In Proc. 6th International Workshop on Automation of Software Test*, AST '11, ACM: New York, NY, USA, 2011; 1–7, doi:10.1145/1982595.1982597.
21. Chen TY, Ho JWK, Liu H, Xie X. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics* 2009; **10**.
22. Guderlei R, Mayer J. Statistical metamorphic testing – testing programs with random output by means of statistical hypothesis tests and metamorphic testing. *In Proc. 7th International Conference on Quality Software (QSIC)*, 2007;

Copyright © 2015 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

404 –409, doi:10.1109/QSIC.2007.4385527.

23. Chen TY, Feng J, Tse TH. Metamorphic testing of programs on partial differential equations: A case study. *In Proc. 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, IEEE Computer Society: Washington, DC, USA, 2002; 327–333.
24. Segura S, Durn A, Snchez AB, Berre DL, Lonca E, Ruiz-Corts A. Automated metamorphic testing of variability analysis tools. *Software Testing, Verification and Reliability* 2015; **25**(2):138–163, doi:10.1002/stvr.1566. URL http://dx.doi.org/10.1002/stvr.1566.
25. Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. *In Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM: New York, NY, USA, 2014; 216–226, doi:10.1145/2594291.2594334. URL http://doi.acm.org/10.1145/2594291.2594334.
26. Nez A, Hierons R. A methodology for validating cloud models using metamorphic testing. *Annals of Telecommunications* 2015; **70**(3-4):127–135, doi:10.1007/s12243-014-0442-7. URL http://dx.doi.org/10.1007/s12243-014-0442-7.
27. Lindvall M, Ganesan D, Ardal R, Wiegand RE. Metamorphic model-based testing applied on NASA DAT – an experience report. *In Proc. 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, vol. 2, 2015; 129–138, doi:10.1109/ICSE.2015.348.
28. Allen FE. Control flow analysis. *SIGPLAN Not.* Jul 1970; **5**(7):1–19, doi:10.1145/390013.808479. URL http://doi.acm.org/10.1145/390013.808479.
29. Vallee-Rai R, Hendren LJ. Jimple: Simplifying Java bytecode for analyses and transformations 1998.
30. Kondor RI, Lafferty J. Diffusion kernels on graphs and other discrete structures. *In Proc. 19th International Conf. on Machine Learning*, 2002; 315–322.
31. Gärtner T, Flach P, Wrobel S. On graph kernels: Hardness results and efficient alternatives. *Learning Theory and Kernel Machines*, *Lecture Notes in Computer Science*, vol. 2777, Schölkopf B, Warmuth M (eds.). Springer Berlin Heidelberg, 2003; 129–143, doi:10.1007/978-3-540-45167-9_11.
32. Shervashidze N, Vishwanathan SVN, Petri T, Mehlhorn K, Borgwardt K. Efficient graphlet kernels for large graph comparison. *In Proceedings of the International Workshop on Artificial Intelligence and Statistics*, Welling M, van Dyk D (eds.), Society for Artificial Intelligence and Statistics, 2009.
33. Murphy C, Kaiser GE, Hu L, Wu L. Properties of machine learning applications for use in metamorphic testing. *In Proc. 20th International Conference on Software Engineering & Knowledge Engineering (SEKE'2008)*, San Francisco, CA, USA, 2008; 867–872.
34. Hu P, Zhang Z, Chan WK, Tse TH. An empirical comparison between direct and indirect test result checking approaches. *Proceedings of the 3rd International Workshop on Software Quality Assurance*, SOQUA '06, ACM: New York, NY, USA, 2006; 6–13, doi:10.1145/1188895.1188901. URL http://doi.acm.org/10.1145/1188895.1188901.
35. Liu H, Kuo FC, Towey D, Chen TY. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering* Jan 2014; **40**(1):4–22, doi:10.1109/TSE.2013.46.
36. Mishra KS, Kaiser G. Effectiveness of teaching metamorphic testing. *Technical Report CUCS-020-12*, Department of Computer Science, Columbia University 2012.
37. Ben-Hur A, Weston J. A User's Guide to Support Vector Machines. *Data Mining Techniques for the Life Sciences*, *Methods in Molecular Biology*, vol. 609, Carugo O, Eisenhaber F (eds.). chap. 13, Humana Press: Totowa, NJ, 2010; 223–239, doi:10.1007/978-1-60327-241-4_13. URL http://dx.doi.org/10.1007/978-1-60327-241-4_13.
38. Huang J, Ling C. Using AUC and accuracy in evaluating learning algorithms. *IEEE Transactions on Knowledge and Data Engineering* Mar 2005; **17**(3):299 – 310, doi:10.1109/TKDE.2005.50.
39. Murphy C. Metamorphic testing techniques to detect defects in applications without test oracles. PhD dissertation, Columbia University 2010.
40. DeMillo R, Lipton R, Sayward F. Hints on test data selection: Help for the practicing programmer. *Computer* Apr 1978; **11**(4):34 –41, doi:10.1109/C-M.1978.218136.
41. Murphy C, Shen K, Kaiser G. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. *In Proc. 2009 International Conference on Software Testing Verification and Validation*, ICST '09, IEEE Computer Society: Washington, DC, USA, 2009; 436–445, doi:10.1109/ICST.2009.19.
42. Murphy C, Shen K, Kaiser G. Automatic system testing of programs without test oracles. *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, ACM: New York, NY, USA, 2009; 189–200, doi:10.1145/1572272.1572295.
43. Briand LC. Novel applications of machine learning in software testing. *In Proc. 8th International Conference on Quality Software*, IEEE Computer Society: Washington, DC, USA, 2008; 3–10, doi:10.1109/QSIC.2008.29.

Copyright © 2015 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

44. Bowring JF, Rehg JM, Harrold MJ. Active learning for automatic classification of software behavior. *In Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, ACM: New York, NY, USA, 2004; 195–205, doi:10.1145/1007512.1007539.
45. Briand LC, Labiche Y, Bawar Z. Using machine learning to refine black-box test specifications and test suites. *In. Proc 8th International Conference on Quality Software*, QSIC '08, IEEE Computer Society: Washington, DC, USA, 2008; 135–144, doi:10.1109/QSIC.2008.5.
46. Briand LC, Labiche Y, Liu X. Using machine learning to support debugging with tarantula. *In Proc. 18th IEEE International Symposium on Software Reliability (ISSRE)*, 2007; 137 –146, doi:10.1109/ISSRE.2007.31.
47. Frounchi K, Briand LC, Grady L, Labiche Y, Subramanyan R. Automating image segmentation verification and validation by learning test oracles. *Inf. Softw. Technol.* Dec 2011; **53**(12):1337–1348, doi:10.1016/j.infsof.2011.06.009.
48. Lo JH. Predicting software reliability with support vector machines. *In Proc. 2nd International Conference on Computer Research and Development*, 2010; 765 –769, doi:10.1109/ICCRD.2010.144.
49. Wang F, Yao LW, Wu JH. Intelligent test oracle construction for reactive systems without explicit specifications. *In Proc. 9th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, 2011; 89 –96, doi:10.1109/DASC.2011.39.
50. Evgeniou T, Pontil M. Regularized multi-task learning. *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2004; 109–117.
51. Madjarov G, Kocev D, Gjorgjevikj D, Džeroski S. An extensive experimental comparison of methods for multi-label learning. *Pattern Recognition* 2012; **45**(9):3084 – 3104, doi:http://dx.doi.org/10.1016/j.patcog.2012.03.004. URL http://www.sciencedirect.com/science/article/pii/S0031320312001203.
52. Shawe-Taylor J, Cristianini N. *Kernel Methods for Pattern Analysis*. Cambridge University Press: New York, NY, USA, 2004.
53. Borgwardt K, Kriegel HP. Shortest-path kernels on graphs. *In Proc. 5th IEEE International Conference on Data Mining*, 2005; 74–81, doi:10.1109/ICDM.2005.132.
54. Ramon J, Gärtner T. Expressivity versus efficiency of graph kernels. *In Proc. 1st International Workshop on Mining Graphs, Trees and Sequences*, 2003; 65–74.
55. Borgwardt KM, Ong CS, Schnauer S, Vishwanathan SVN, Smola AJ, Kriegel HP. Protein function prediction via graph kernels. *Bioinformatics* 2005; **21**(suppl 1):i47–i56, doi:10.1093/bioinformatics/bti1007.

Copyright © 2015 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls