USING SEMI-SUPERVISED LEARNING

FOR PREDICTING METAMORPHIC RELATIONS

by

Bonnie Elizabeth Hardin

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

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2018

ACKNOWLEDGEMENTS

I would like to thank several people for their help in finishing this project. First, my advisor Dr. Upulee Kanewala for her advice and guidance over the past few years. I would also like to thank my classmates Safia, Prashanta, Karishma, and Madhu who have worked along side me and encouraged me, and my husband for his support. Finally, I would like to thank Zoot Enterprises for their financial support.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

Software testing is difficult to automate, especially in programs which face the oracle problem, where an oracle does not exist, or is too hard to develop. Metamorphic testing is a solution to this problem. Metamorphic testing uses metamorphic relations to determine if tests pass or fail. A large amount of time is needed for a domain expert to determine which metamorphic relations can be used to test a given program. Metamorphic relation prediction removes this need for such an expert. We propose a method using semi-supervised learning algorithms to detect which metamorphic relations are applicable to a given code base. Semi-supervised learning is useful in this problem domain as most programs do not have pre-defined metamorphic relations. These programs are considered unlabeled data in a semi-supervised algorithm. We compare two semi-supervised models with a supervised model, and show that the addition of unlabeled data improves the classification accuracy of the metamorphic relation prediction model.

1

INTRODUCTION

With the rapid growth of science and technology and the role it plays in the world, it is increasingly necessary to verify the accuracy of the software that produces new scientific findings. Researchers in all scientific domains face a difficult problem when it comes to testing their software. Generally, the correctness of software is determined by comparing the results of the program with the expected results. However, in the case of scientific code, the correct results are often not known; this complication, known as the oracle problem, makes testing scientific software a difficult task [3].

Metamorphic testing (MT) is one solution to the oracle problem. MT requires the usage of metamorphic relations (MRs) to act as an oracle for the program under test; an MR defines how a change to a test input will change the corresponding outputs. Defining MRs must often be done by the scientific domain expert, and is a time-consuming process. The greater the cost needed to built a test suite, the less likely a company or researcher is to use it. Therefore, MR prediction models are needed to decrease the amount of time and cost needed to construct an MT suite.

There are three main categories of machine learning methods: supervised, semi-supervised, and unsupervised. In supervised machine learning algorithms, all of the data used to build the classifier have labels. These models can be unpractical because of the cost of obtaining the initial labels. In semi-supervised models, the majority of the data is unlabeled. These models all becoming increasingly necessary, as big data is easily accessible on the Internet, but the corresponding labels are not. The structure of the unlabeled data helps to provide greater classification accuracy than

with labeled data alone.

Previous studies have shown that supervised learning algorithms, including support vector machines and decision trees, are effective for predicting MRs [8, 9]. Our current study extends these works to include unlabeled data in the training of the binary classifiers. Unlabeled data has been shown to increase classification accuracy of machine learning models [15]. Additionally, there are many methods that do not have pre-determined metamorphic relations. These unlabeled methods can easily be added to a semi-supervised model to increase classification accuracy.

In this work we use two semi-supervised binary classification algorithms, semi-supervised support vector machine and label propagation, to predict MRs for a given program. For label propagation, we use explicit features extracted from control flow graphs. For the semi-supervised support vector machine, we use graph kernels.

Our results show that the label propagation algorithm performs better than the support vector machine for 5 out of the 6 studied MRs. The semi-supervised support vector machine performs better than the support vector machine for 4 out of the 6 MRs. These results suggest the conclusion that the addition of unlabeled data, in a semi-supervised algorithm, can improve on the classification accuracy of a supervised machine learning model for MR predictions.

The rest of this document is organized as follows. In chapter 2 we discuss the relevant background information, including an overview of metamorphic testing, metamorphic relations, and semi-supervised learning algorithms. We also highlight the most interesting recent work in the field of metamorphic relation creation. In chapter 3 we present the method, using label propagation, support vector machines, and semi-supervised support vector machines. In chapter 4, we discuss the evaluation measures used against the three models built, and present the results. Finally, in chapter 5, we give a conclusion.

3

BACKGROUND

Oracles

The oracle problem described in Chapter 1 is common in scientific code, as the code is used to calculate previously unknown research findings. The human oracle cost, or time needed for a test engineer to manually define inputs and expected outputs, can be a large burden on the software testing process [12]. Therefore, software testing where no oracle is present is a focus of many current studies. Metamorphic testing (MT) is one of these such studies which can ease the burden on a testing budget caused by the oracle problem.

Metamorphic Testing

Metamorphic testing allows for correct outputs to be verified in programs that have no readily available oracle. MT requires the use of Metamorphic Relations (MR). An MR is a relationship between an input and an output of a program. Every MR acts as an oracle for the Program Under Test (PUT); it defines how a change to a test input will affect the corresponding output. If the output is not changed in the pre-defined way, given that the MR is correctly defined, the program must contain a fault.

Metamorphic testing follows this process:

1. Select an MR that applies to the PUT. This often must be accomplished by a domain expert.

2. Generate an initial test case using any kind of test generation technique.

3. Generate the follow-up test case by transforming the initial test case based on the chosen MR.

4. Run both test cases.

5. Compare the outputs. The initial test result and follow-up test result should be related in a specific way based on the chosen MR.

As an example, consider the code snippet in Figure 2.1, and its corresponding metamorphic test in Table 2.1. First, we identify the Multiplicative MR will apply to this method. Second, we manually generate an initial test case, consisting of the input $a = [1, 5, 2]$ and the expected change to the output (order stays the same). Next, we use the chosen MR to transform the initial test case into the follow-up test case. We do so by multiplying each element of the array by a randomly chosen constant, 2, yielding the follow-up test $a' = [2, 10, 4]$. After identifying that multiplying the elements of a by a constant will produce a reliable change in the output, the next step is to define what specific change is expected. Given that the first test case returns $r = [a, b, c]$ as the result, we expect the follow-up test case to return $r' = [a * 2, b * 2, c * 2]$. If the two outputs do not satisfy the expected relationship, we know there is an error in the code. On the contrary, if the outputs do satisfy the expected relationship, we cannot say the code is error-free, only that this one test case has passed. Any type of software testing cannot prove a piece of code is completely error unless all possible paths of execution through the code are tested.

Initial test case	$a = [1, 5, 2]$	$r = [a, b, c]$	Initial output
Follow-up test case	$a' = [2, 10, 4]$	$r' = [a * 2, b * 2, c * 2]$	Follow-up output

Table 2.1: Metamorphic test for bubble sort

```
public static  int[] bubble( int[] a )
{
    int i;
    int j;
    int t;
    for (i = a.length - 2; i >= 0; i--) {
        for (j = 0; j <= i; j++) {
            if (a[j] > a[j + 1]) {
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
    }
    return a;
}
```

Figure 2.1: A method that performs bubble sort

Machine Learning

Machine learning is a way for programs to classify data based on previously seen information. There are three main categories of machine learning methods: supervised, semi-supervised, and unsupervised [1].

In supervised machine learning algorithms, all of the data used to build the classifier have labels [10]. More formally, a supervised machine learning model consists of a set of elements to be classified, $S = \{s_1, s_2, ..., s_n\}$, a set of features or defining

characteristics for each s_i, $X = \{x_1, x_2, ..., x_n\}$, and a list of labels $L = \{l_1, l_2, ..., l_n\}$ where each l_i corresponds to the i^{th} element of S. In contrast, unsupervised machine learning models use the element set S and the feature set X, but do not have any labels associated with the elements to be classified. In semi-supervised models, the majority of the data is unlabeled. This type of model is becoming increasingly necessary, as big data is easily accessible on the Internet, but the corresponding labels are not. Semi-supervised models use a set of labeled elements to be classified, X, a set of unlabeled elements to be classified, Y, and a set of features, S. The labels list L only contains the labels for the elements in X [17].

Machine learning algorithms can also be categorized into classification problems and regression problems. Classification algorithms apply a binary label to the unseen data. Regression algorithms apply a continuous value to the unseen data. This work uses semi-supervised, classification algorithms to apply a binary label of "MR applies" or "MR does not apply" to each method, for each studied metamorphic relation.

In the following paragraphs, we describe the three algorithms used in the method, label propagation, support vector machine, and semi-supervised support vector machine.

Label Propagation

Label propagation, a semi-supervised learning algorithm, was first proposed in 2002 by Zhu et al [18]. This algorithm, which requires the data to be represented as a connected graph, fixes the labels for the originally labeled points, and propagates those labels to unlabeled points based on graph proximity.

The problem definition is as follows. Let $X_l = \{(x_1, y_1), ..., (x_l, y_l)\}$ be the set of labeled data points. $X_u = \{(x_{l+1}, y_{l+1}), ..., (x_{l+u}, y_{l+u})\}$ is the set of unlabeled data points. Y_l is the list of known labels, and Y_u is the list of unknown labels. Given an

input of X_u, X_l, Y_l, label propagation builds a model that can predict labels for a set of previously unseen data points.

The algorithm consists of three steps:

1. propagate $Y \leftarrow TY$
2. row normalize Y
3. clamp the labeled data

T is a probabilistic transition matrix, where T_{ij} is the probability of jumping from node j to node i. These probabilities are extracted from the feature set. The value for any given T_{ij} is calculated as $w_{ij} / \sum_{k=1}^{l+u} w_{kj}$, where w_{ij} is a predetermined weight directly correlated with the Euclidean distance of the two nodes i and j.

Y is a label matrix that represents the label probability distributions of each data point. The dimensions are $(l + u)$ x C, where l is the length of the labeled data, u is the length of the unlabeled data, and C is the set of labels. Y_i contains the probability that i is assigned to each label $c \in C$.

The first step in the algorithm propagates the labels to previously unlabeled points. Then, Y is row-normalized to ensure the label probabilities retain their meanings. The labeled data is clamped in the third step of the algorithm. This step is to ensure the original labels do not change.

Semi-Supervised Support Vector Machine

Before introducing a Semi-Supervised Support Vector Machine (S3VM), we explain the Support Vector Machine (SVM). An SVM requires the data to be represented as Cartesian coordinates. Throughout the course of the algorithm, the data points are separated into two sections, with the same label being assigned to all points in a section [5].

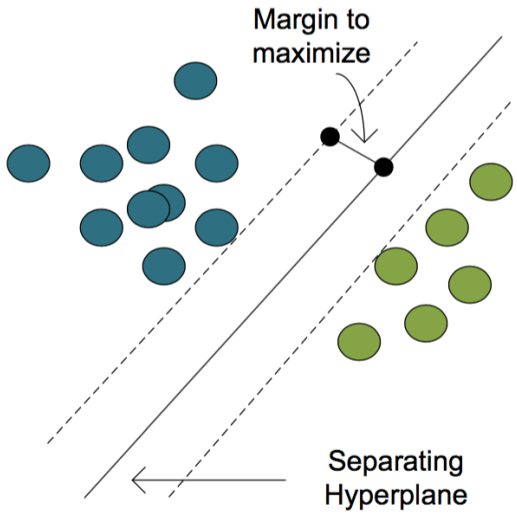

Figure 2.2: Support Vector Machine

The algorithm finds a hyperplane among the data points that separates both classes of data. As shown in Figure 2.2, a hyperplane is chosen such that the margin between the hyperplane and the support vectors is maximized. This choice allows for a greater separation of the classes, thereby decreasing classification error. The optimal hyperplane is calculated using either the primal or the dual problem.

Machine learning models can be transductive or inductive. Transductive models make predictions for testing sets based on specific data patterns seen in training sets. Inductive models on the other hand, make predictions for testing sets based on mathematical models derived from the training sets. An SVM is an inductive model, as it builds a classifier from labeled data. This classifier, or learned function, can be used to predict labels for previously unseen, unlabeled data. Label Propagation is a transductive model, as it applies labels to the unlabeled data points based on the known data.

The concept of a semi-supervised support vector machine was first introduced in 1999 by Bennett and Demiriz [4]. An S3VM is similar to an SVM, but differs

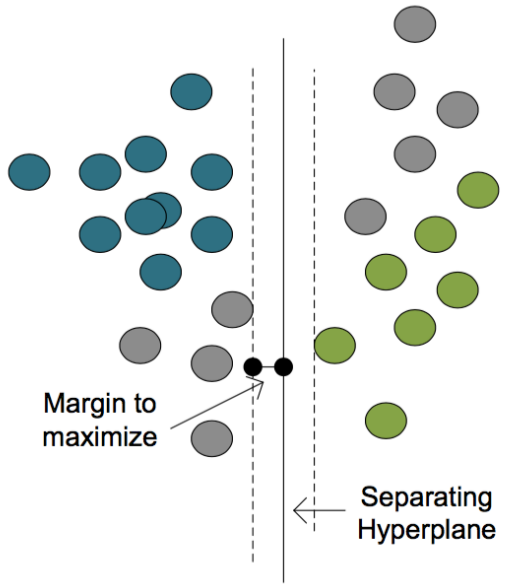

Figure 2.3: Semi-Supervised Support Vector Machine

in two key areas. First, the S3VM accepts unlabeled data, while in the SVM all data must be labeled. Second, an S3VM is a transductive model. In some literature, an S3VM is also referred to as a Transductive Support Vector Machine. The S3VM algorithm produces a prediction function that will predict labels for previously unseen data points. Two additional constraints are added to the optimization problem in an S3VM. These constraints calculate the misclassification errors for each data point and class. The algorithm chooses the class with the smallest misclassification error. The addition of unlabeled data to an S3VM, shown in Figure 2.3, can improve the classification accuracy of the model.

Evaluation Methods for Machine Learning

There are several common metrics for evaluating performance of a model. Accuracy is the percentage of correct predictions. While this is a commonly used classifier evaluation metric, it has several flaws. First, it does not consider the

	Predicted label: 0	Predicted label: 1
True label: 0	True negative	False positive
True label: 1	False negative	True positive

Table 2.2: Confusion matrix

Metric	Description	Calculation
accuracy	percentage of correct predictions	$(TP + TN)/total$
misclassification rate	percentage of incorrect predictions	$(FP + FN)/total$
precision	percentage of positive predictions that are correct	$TP/(FP + TP)$
sensitivity	percentage of correct predictions for the true positive class	$TP/(FN + TP)$

Table 2.3: Common classifier evaluation metrics calculated from a confusion matrix

distribution of the class labels. A model trained with 90% of the data in the positive class, and 10% of the data in the negative class, may result in a high accuracy score by simply predicting 1 for most of the data instances. Accuracy also does not show the types of errors a model is making. Null accuracy is often calculated along with accuracy to make the accuracy score more meaningful. Null accuracy is the accuracy that is achieved by always predicting the most common class. Precision represents the percentage of predictions for the positive class that are correct. This metric is useful when it is most important for the model to identify positives than negatives, and false positives are not a large concern. A confusion matrix is a common metric used in model evaluation, shown in Table 2.2. Confusion matrices show all prediction outcomes, which gives the user a more complete view of the correctness of the model. Several other useful metrics can be calculated from a confusion matrix. These are shown in Table 2.3.

Area under the receiver operating characteristic (AUROC), is also commonly

referred to as area under the ROC curve. AUROC is considered to be a more useful metric in general than accuracy or precision. This metric ranges from 0 to 1, with 1 representing a perfect classifier. A score of 0 indicates the classifier would be perfect if inverted. To calculate AUROC, the false positives rates are plotted on the x axis and the true positive rates are plotted on the y axis for each threshold from 0 to 1. Threshold here means the separating value at which a data point is classified as either the positive or negative class. The ROC curve is the line created by the plotted false positives and true positives for each threshold. The score, AUROC, is calculated as the area under the ROC curve.

Cross validation is another important feature in evaluating a model. It does not provide a direct score for the model, but shows approximately how a model would perform on an unseen data set. K-fold cross validation is a common method used to evaluate a model during the training phase. While k is commonly chosen to be 10, any integer value could be used. The first step in k-fold cross validation is to split the training data into k partitions. Of these k partitions, $k-1$ are used to train the model, and the remaining partition is used as a validation set to test the accuracy of the model. This cycle is repeated for a total of k times, so that each time the validation set is a different one of the k partitions. In this way, each validation set is used once, and each combination of $k-1$ partitions is used as the training set. The accuracy (or other metric) scores are averaged to produce an approximation of how well the model will perform against a third, previously unseen, testing set. Our method uses 10-fold cross validation for the purpose of choosing optimal parameters for the label propagation and support vector machine models.

Stratified cross validation is a means of keeping the same proportion of classes in each split. If a full data set consists of 70% positive instances and 30% negative, stratified cross validation would split the data set such that each split retains the

original 70/30 proportion. Our method uses stratified cross validation to ensure the proportions of positive and negative data points in the full data set are maintained within the training/validation/testing sets.

Control Flow Graphs

A control flow graph (CFG) is a graph representation of source code, consisting of a set of nodes and edges [2]. The nodes represent the basic blocks in a program, and the edges represent the paths between the basic blocks. A basic block is a set of executable statements that have only one entry point and one exit point. Consider the code in Figure 2.4 and its corresponding CFG. Each basic block in the code has a corresponding node in the graph.

More formally, a CFG is a directed graph $G = (V, E)$, where V is the set of vertices in the graph, and E is the set of edges in the graph. Each vertex $v \in V$ represents an executable statement in the code. Each edge $e \in E = (v_x, v_y)$ if x, y are executable statements in the code where y is executed directly following x. These graphs can be used to mine interesting information about programs such as structure, complexity, and testing capabilities.

There are many variations of a CFG, including a Simplified Control Flow Graph, Call Graph, Dependency Graph, and Super Control Flow Graph. A Simplified Control Flow Graph ignore basic blocks are stores one node for each executable statement. A Call Graph has one node for each method, and an edge for each function call. A dependency graph has one node for each executable statement, and an edge for each for each data dependency in the code. A super control flow graph has all the properties of a CFG, with the addition of edges between function call locations and function call returns.

```
public static int my_method(int a)
{
   a = a + 1;
   if(a < 5){
      result = 1;
   }
   result = result + 1;
}
```

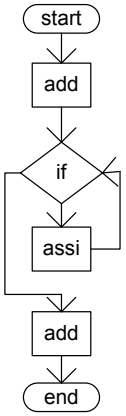

Figure 2.4: Java method and its corresponding control flow graph

Related Work

In the field of metamorphic relation development, several other studies have taken place recently.

A 2012 study by Liu et al. proposes a method for the composition of MRs [11]. Their study showed that by combining two or more MRs, they can produce a new MR with a higher fault-finding effectiveness than the original. Two MRs are "compositable" if for any source test case t_1 and its MR m_1, the corresponding follow-up test case t_2 can be used as a source test case for a second MR, m_2. So t_1 and t_2 are "compositable", thereby creating a new MR.

A second paper by Su et al. studies the dynamic inference of MRs [16]. The authors built a tool to implement their algorithm. The algorithm works by first defining a set of MRs, which they call transformers: "multiplier", "adder", "negator", "shuffler", and "reverser". For each transformer in the set, a function is executed with and without the transformation applied. The results are compared to see if the functions exhibit a metamorphic property. In this way, they can predict which MRs apply to a given function.

The final paper, and most similar to this study, that we mention is a 2013 study by Kanewala et al [8]. This study has many similarities with our current work in that it uses a feature set consisting of node and path data through a function's control flow graph. These features are input into an SVM and a decision tree to build binary classifiers for metamorphic relations. The key difference between this study and ours is that this study uses supervised learning techniques, while ours extends into semi-supervised learning. Semi-supervised learning classifiers can be more accurate than their supervised counterparts because of the addition of unlabeled data [15].

15

METHOD

In this chapter, we present our method for predicting MRs using semi-supervised learning. The overview of the method is shown in Figure 3.1. The goal of the method is to determine if semi-supervised learning can perform as well as or better than supervised learning for metamorphic relation classification. Our data set consists of Java programs, which we transform into their control flow graphs. From these graphs, we extract both a feature set and two calculated graph kernels. Both the features and the kernels are input to the selected machine learning algorithms to train a binary classification model. These models are then used to predict labels for new methods.

Feature Extraction

Based on the results of previous studies [8] [9], we believe there is a correlation between the paths taken through a method and the metamorphic relations found in Table 4.2. MRs are defined directly based on the sequence of operations performed in a program; therefore, there must be a relationship between the paths taken through a method and the MRs which are suitable tests. To evaluate this hypothesis, we transform a set of Java methods into their control flow graphs, and use elements of the CFGs as features for our machine learning models. We utilized two types of

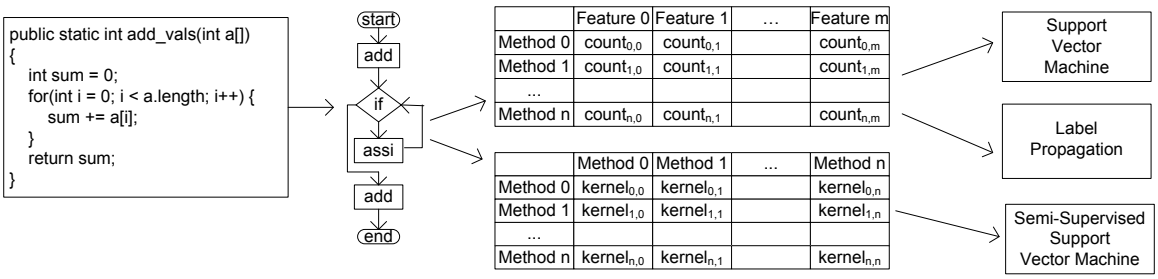

Figure 3.1: Overview of Method

Feature	Feature Count
start - 0 - 1	1
add - 1 - 1	1
if - 2 - 2	1
assi - 1 - 1	1
add - 1 - 1	1
end - 1 - 0	1

Table 3.1: Node features for the CFG in Figure 3.1

features: node and path features.

A node feature consists of the given node, followed by the in degree and the out degree. The node features for the CFG in Figure 3.1 are shown in Table 3.1. Path features consist of the shortest paths from the start node to each node in the graph, and the shortest paths from each node in the graph to the end node. The path features for the CFG in Figure 3.1 are shown in Table 3.2. Paths through a control flow graph represent the possible executions of a program. Similarly, a metamorphic relation is chosen for a program based on the possible paths of execution. Therefore, we believe a feature set consisting of paths through a program is a good predictor for metamorphic relations.

Graph Kernels

In many binary data sets, the classes have no clear linear separation. Adding in extra dimensions often creates a separating hyperplane that was not possible in two dimensions. Kernels are similarity functions that allow separating hyperplanes to be more easily found without processing a model in higher dimensions. Many algorithms

Feature	Feature Count
start - add	1
start - add - if	1
start - add - if - assi	1
start - add - if - add	1
start - add - if - add - end	1
add - if - add - end	1
if - add - end	1
assi - if - add - end	1
add - end	1

Table 3.2: Path features for the CFG in Figure 3.1

use a linear kernel, which is calculated using the dot product. We use two pre-defined kernels, random walk and graphlet. These kernels are more specific to our data set than the commonly used linear or RBF kernels.

The random walk kernel follows paths up to a certain length in each graph, then evaluates the amount of similar or identical paths among two graphs. For the two graphs in Figure 3.2, we show the random walk kernel computation process in Table 3.3.

The graphlet kernel shares some similarities with the random walk kernel, namely the node and edge similarity metric. The main difference is that the graphlet kernel considers graph structures for each subgraph, or graphlet. We consider subgraphs with a vertex count of 3, 4, and 5. For the graphs in Figure 3.3, we show the graphlet kernel calculation process in Table 3.4. First the subgraphs, or graphlets, of lengths 3, 4, and 5 are extracted from the main graphs. If a pair of graphlets are isomorphic,

Similarity score between two walks: $k_{walk}(A->B, D->E) = k_{step}((A,B),(D,E))$ ... $k_{walk}(A->B->C, D->E->F) = k_{step}((A,B),(D,E))$ x $k_{step}((B,C),(E,F))$
Similarity score between two steps: $k_{step}((A,B),(D,E)) = k_{node}(A,D)$ x $k_{node}(B,E)$ x $k_{edge}((A,B),(P,Q))$ $k_{step}((A,B),(D,E)) = k_{node}(A,D)$ x $k_{node}(B,E)$ x $k_{edge}((A,B),(D,E))$ $k_{step}((B,C),(E,F)) = k_{node}(B,E)$ x $k_{node}(C,F)$ x $k_{edge}((A,B),(E,F))$
Similarity score between two nodes: $k_{node}(A,D) = 0.5$ (similiar properties between nodes) $k_{node}(B,E) = 1$ (identical nodes) $k_{node}(B,F) = 0$ (dissimilar properties between nodes)
Similarity score between two edges: $k_{edge}((A,B),(D,E)) = 1$ (identical labels)

Table 3.3: Random walk kernel calculation method

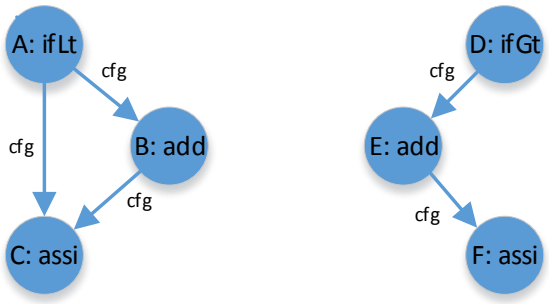

Figure 3.2: Graphs for random walk kernel calculation in Table 3.3.

meaning they contain the same number of vertices with the same edge structures, the kernel value is calculated using the node and edge scores defined in the random walk kernel. If the pair are of graphlets not isomorphic, the kernel value is set to zero.

Baseline Approach

As a baseline approach, we use a support vector machine. This supervised model was chosen as a baseline because it performed well in previous studies [8]. Our study aims to determine if semi-supervised learning can be as effective or better than supervised models. Therefore, we compare the two selected semi-supervised models to the SVM to evaluate our hypothesis.

To build the SVM model, we use scikit-learn, which is a collection of Python machine learning modules [14]. We selected scikit-learn's LinearSVC implementation of an SVM. The code for this model is found on our software testing lab GitHub page[1]. We built three variations of this model. The first variation is using the feature set extracted from the CFGs. This model is compared to the label propagation model, which uses the same feature set. The second and third variations are using the random

[1]https://github.com/MSU-STLab/MRPrediction/svm.py

Similarity score between two graphs using graphlets: $k_{graphlet}(G1, G2) = k_{subgraph}(G1.1, G2.1) + k_{subgraph}(G1.1, G2.2) + ...+$ $k_{subgraph}(G1.2, G2.2) + k_{subgraph}(G1.2, G2.3)$
Similarity score between two graphlets: $k_{subgraph}(G1.1, G2.1) = 0$ $k_{subgraph}(G1.1, G2.2) = 0$ $k_{subgraph}(G1.2, G2.2) = k_{node}(A, E)$ x $k_{node}(B, F)$ x $k_{node}(D, H)$ x $k_{edge}((A, B), (E, F))$ x $k_{edge}((B, D), (F, H))$ ... $k_{subgraph}(G1.2, G2.3) = 0$
Similarity score between two nodes: $k_{node}(A, E) = 0$ $k_{node}(B, F) = 0$ ... $k_{node}(D, H) = 0$
Similarity score between two edges: $k_{edge}(A, B) = 1$ ... $k_{edge}(F, H) = 1$

Table 3.4: Graphlet kernel calculation method

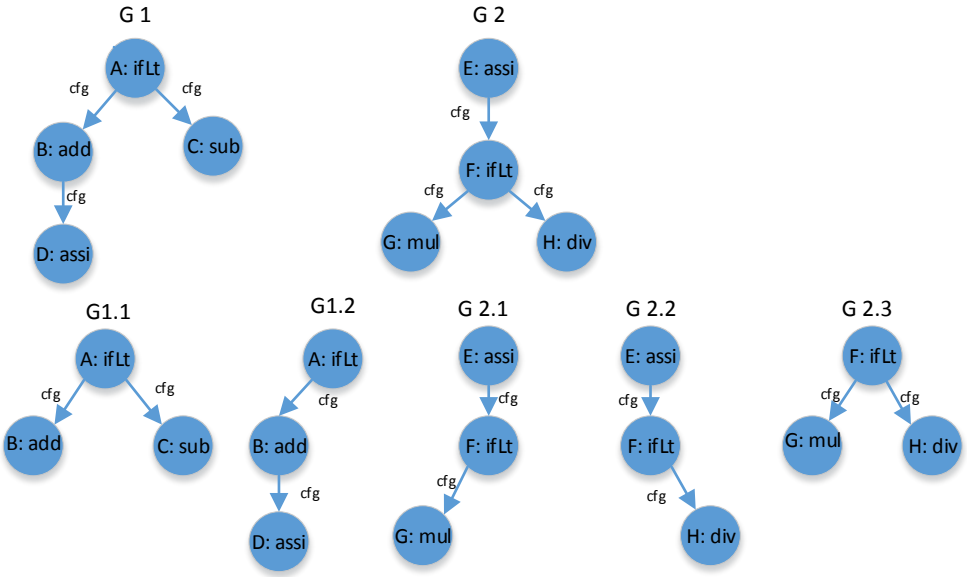

Figure 3.3: Graphs for graphlet kernel calculation in Table 3.4.

walk kernel and graphlet kernel. These models are compared to the S3VMs with the same kernels.

Label Propagation

Our first semi-supervised model is label propagation. Scikit-learn implements two semi-supervised algorithms: label propagation and label spreading. We chose label propagation because it clamps the original true labels; label spreading allows for the input label distributions to change over the course of the algorithm. Because our data set is relatively small, we believe clamping the known labels will yield higher classification accuracy. This model is built using the CFG feature set, and compared to the SVM built with the same features. The code for this model is found on our

software testing lab GitHub page[2].

Semi-Supervised Support Vector Machine

Our second semi-supervised model is a semi-supervised support vector machine. We choose to use an implementation called QN-S3VM, designed by Gieseke et al. [6]. We selected this tool over other S3VMs because of its ability to handle various types of kernels. This tool contained built-in code for using linear and RBF kernels. We modified the code to read in our precomputed kernels (random walk and graphlet). The code for this model can be found on our software testing lab GitHub page[3]. We compared this model to the SVM that was also built with the random walk and graphlet kernels. The labeled data in the SVM and S3VM are the same data points; the S3VM contains additional unlabeled data points.

[2]https://github.com/MSU-STLab/MRPrediction/labelPropagation.py
[3]https://github.com/MSU-STLab/MRPrediction/s3vm.py

EVALUATION AND RESULTS

In this chapter, we discuss the evaluation methods used for each of the models built, and present the results.

Evaluation Method

We begin with a collection of 100 open source Java methods. These methods all perform actions related to scientific computing, such as sorting or calculating the Hamming distance. This collection of methods was chosen as the subject of our study because of the common actions they perform in scientific computing. The methods used in our study are shown in Table 4.1.

For the algorithms that use a feature set, we use a python module that uses a depth-first search to generate all the simple paths found in a graph. The running time of this module is $O(n!)$, given a complete graph with n nodes. Therefore, we removed graphs with greater than 30 vertices from the set, leaving a total of 62 methods used in the three algorithms.

Method	**Open source project**	**Lines of code**
min	Colt Project	10
max	Colt Project	10
covariance	Colt Project	14
durbinWatson	Colt Project	9
meanDeviation	Colt Project	8
product	Colt Project	8
weightedMean	Colt Project	11
autoCorrelation	Colt Project	8

Method	**Open source project**	**Lines of code**
quantile	Colt Project	16
sumOfLogarithms	Colt Project	7
sampleVariance	Colt Project	9
weightedRMS	Colt Project	9
harmonicMean	Colt Project	7
sumOfPowerofDeviations	Colt Project	7
power	Colt Project	6
square	Colt Project	6
polevl	Colt Project	9
cosineDistance	Apache Mahout	20
manhattanDistance	Apache Mahout	7
chebyshevDistance	Apache Mahout	14
hammingDistance	Apache Mahout	1
sum	Apache Mahout	1
errorRate	Apache Mahout	1
scale	Apache Math	7
distance1	Apache Math	7
distanceInf	Apache Math	7
ebeAdd	Apache Math	10
ebeDivide	Apache Math	10
ebeMultiply	Apache Math	10
entropy	Apache Math	14
calculateAbsoluteDifferences	Apache Math	13
computeCanberraDistance	Apache Math	9

Method	**Open source project**	**Lines of code**
evaluateHoners	Apache Math	7
evaluateNewton	Apache Math	8
meanDifference	Apache Math	7
varianceDifference	Apache Math	17
chiSquare	Apache Math	25
evaluateWeightedProduct	Apache Math	10
geometricMean	Collection	7
weightedMean	Collection	9
dotProduct	Collection	9
reverse	Collection	9
add_values	Collection	8
bubble_sort	Collection	16
sequential_search	Collection	10
selection_sort	Collection	18
array_calc1	Collection	9
set_min_val	Collection	9
array_copy	Collection	9
find_magnitude	Collection	9
find_max2	Collection	9
insertion_sort	Collection	12
mean_absolute_error	Collection	6
count_k	Collection	9
clip	Collection	15
elementwise_max	Collection	11

Method	**Open source project**	**Lines of code**
elementwise_min	Collection	11
count_non_zeroes	Collection	9
cnt_zeroes	Collection	9

Table 4.1: Code base used in label propagation, SVM, S3VM

These methods were converted to control flow graphs using Soot[1], a Java optimization framework. These representations were stored in .dot files. After obtaining the .dot files, we extract the set of features described below from each method. The feature set consists of two types: node features and path features.

We also collect the labels for each element in our data set. For each MR under consideration, a data point is assigned a label of either 1 or 0, if the MR applies or does not apply to the method. Our experiment uses six MRs, shown in Table 4.2. These MRs were defined in a previous study that used metamorphic relations to test machine learning models [13]. We selected these 6 because they are commonly found in scientific computing applications. For each test case, i is an integer value, and c is a constant that is applied to i to change the value for the follow-up test case. Figure 4.3 shows the percentage of data points in the positive or negative class for each MR studied.

To evaluate the SVM classifier, we use a stratified train/validation/test split. The training and validation sets consist of 80% of the original data set, leaving the testing set with the remaining 20%. The SVM builds a classifier to predict labels for previously unseen data points. An SVM takes a parameter c, which represents the penalty parameter of the error term. We use hyper-parameter optimization to select

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

MR	Initial Test Case	Follow-Up Test Case
Addition	$i_1, i_2, ..., i_n$	$i_1 + c, i_2 + c, ..., i_n + c$
Multiplication	$i_1, i_2, ..., i_n$	$i_1 * c, i_2 * c, ..., i_n * c$
Permutation	$i_1, i_2, ..., i_n$	$i_n, i_1, ..., i_2$
Inclusion	$i_1, i_2, ..., i_n$	$i_1, i_2, ..., i_n, i_{n+1}$
Exclusion	$i_1, i_2, ..., i_n$	$i_1, i_2, ..., i_{n-1}$
Inversion	$i_1, i_2, ..., i_n$	$1/i_1, 1/i_2, ..., 1/i_n$

Table 4.2: Metamorphic Relations Used in the Experiment

MR	Positive Class	Negative Class
Addition	57%	43%
Multiplication	68%	32%
Permutation	34%	66%
Inclusion	33%	67%
Exclusion	31%	69%
Inversion	65%	35%

Table 4.3: Percentage of data points in each class

the best value of c for our model. To do so, we build a model with each value for c, and test the models using the validation set.

To evaluate the label propagation classifier, we also use a stratified train/validation/test split. To find the optimal parameters for the label propagation algorithm, we built a nested hyper-parameter optimization method. The training and validation sets combined consist of 80% of the total data set, and the testing set consists of the remaining 20%. All of the unlabeled data is in the training set, making it 60%

unlabeled data and 40% labeled data. The unlabeled data is chosen randomly, but still ensuring the original proportion of positive and negative classes in the full data set is maintained in the set of unlabeled data. The parameters accepted by the scikit-learn implementation of label propagation are shown in Table 4.4. We test each of the models built with the different parameter combinations using the validation set. We then select the best performing model based on accuracy score. Then, that model is tested using the test data. This process is repeated 10 times, and the scores from each run averaged together.

Parameter	Description	Value
n_neighbors	number of neighbors used for the knn kernel	3
alpha	clamping factor	0
max_iter	max number of iterations allowed	1
tol	threshold to consider the system at steady state	1E-10

Table 4.4: Parameters used in label propagation

To evaluate the S3VM classifier, we split the data into training and testing sets. The training set consists of 80% of the total data set, and the testing set consists of the remaining 20%. The unlabeled data is chosen randomly, but still ensuring the original proportion of positive and negative classes in the full data set is maintained in the set of unlabeled data. The model is built 10 times, each time using the same train/test split as the SVM, with the addition of unlabeled data points to the training set.

Results

The results of the SVM and label propagation comparison are shown in Figure 4.1. We performed our method on the six selected MRs in Table 4.2. For 5 out of the 6 MRs, the accuracy of the semi-supervised label propagation model is better than that of the supervised SVM model.

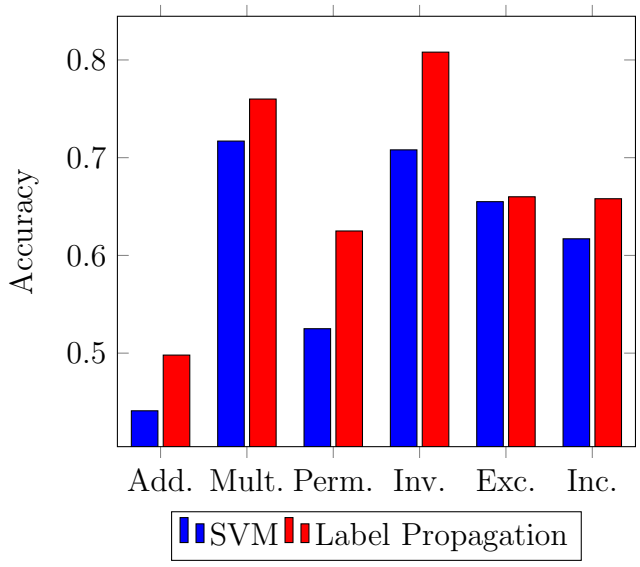

Figure 4.1: SVM and Label Propagation Results

We performed a t-test to determine the statistical relevance of the accuracy improvements. The results comparing SVM and label propagation are shown in Table 4.5. Inversion, Inclusion, and Addition have p-values of less than 0.05, representing a statistically significant change. In a previous supervised learning study, Inversion performed significantly worse than in this study [8]. For this MR, it is clear that the addition of unlabeled data improves the prediction accuracy.

The results of the SVM and S3VM comparisons are shown in Figures 4.2 and 4.3. In 4 out of the 6 MRs using the random walk kernel, the accuracy of the semi-supervised SVM model is better than that of the supervised SVM model. In all 6

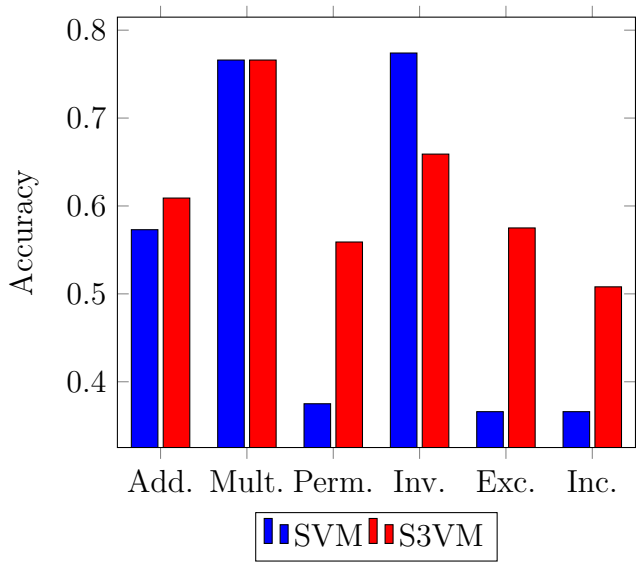

Figure 4.2: SVM and S3VM Results with Random Walk Kernel

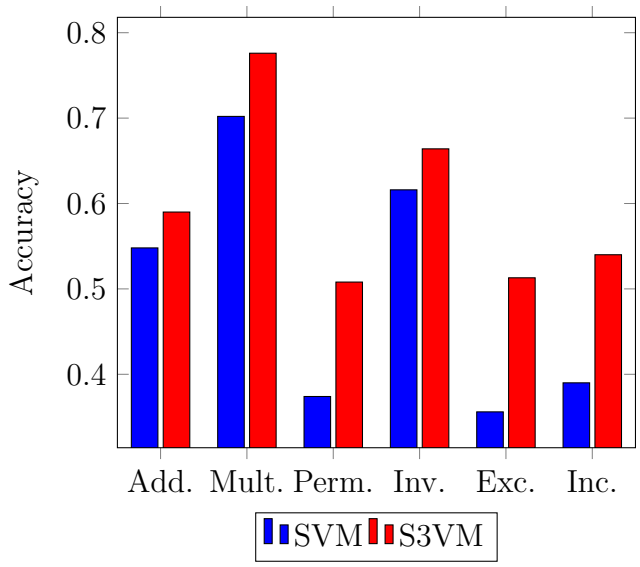

Figure 4.3: SVM and S3VM Results with Graphlet Kernel

MRs using the graphlet kernel, the accuracy of the semi-supervised SVM model is better than that of the supervised SVM model. We performed a t-test to determine the statistical relevance of the accuracy improvements. The results comparing SVM and S3VM are shown in Tables 4.6 and 4.7. For the random walk kernel, Inversion,

MR	p-value
Inversion	0.00362
Inclusion	0.00400
Exclusive	0.34344
Addition	0.05369
Permutation	0.03538
Multiplication	0.12683

Table 4.5: T-test Comparing SVM and Label Propagation.

MR	p-value
Inversion	0.00292
Inclusion	0.00119
Exclusive	0.00070
Addition	0.33463
Permutation	0.00009
Multiplication	1.0000

Table 4.6: T-test Comparing SVM and S3VM Using Random Walk Kernel.

Inclusion, Exclusion, and Permutation have p-values of less than 0.05, representing a statistically significant change. For the graphlet kernel, Exclusion and Permutation have p-values less than 0.05.

For the MRs whose p-values represent a non-significant improvement in accuracy, we believe that the addition of more unlabeled data points would lower the p-values. We believe the small size of our data set is the key reason why these p-values are high. Improving a model with additional unlabeled data is much easier and more practical

MR	p-value
Inversion	0.20619
Inclusion	0.02097
Exclusive	0.00720
Addition	0.26349
Permutation	0.02448
Multiplication	0.21193

Table 4.7: T-test Comparing SVM and S3VM Using Graphlet Kernel.

than to improve one with additional labeled data. For this reason, we believe our method to be a promising approach to use in the future for predicting MRs.

CONCLUSION

In this chapter, we discuss the results and potential threats to the validity of this study, including external and internal validity. We then conclude and present any future work possibilities.

Discussion

The field of software testing is lacking a fast and effective way to test scientific software in an industry setting. Metamorphic testing is a solution to this problem, but requires either manual or automatic creation of metamorphic relations. Previous studies have found methods to predict MRs using supervised learning [8, 9]. These methods require large sets of labeled data, which is difficult to obtain in the field of metamorphic testing.

Our study attempts to fill this gap by using semi-supervised learning models. We evaluate the effectiveness of two semi-supervised models, label propagation and semi-supervised support vector machine, and compare their results to a support vector machine. The key result from our study is that unlabeled data improves metamorphic relation classification accuracy. Our initial findings on this topic were published in the Metamorphic Testing Workshop [7].

Threats to Validity

The main threat to validity for this study is in terms of external validity, or generalizing our results to other situations. The key issue for our study is that of generalizing based on small-scale results. The results of this study suggest semi-supervised learning as an effective class of machine learning algorithms for predicting metamorphic relations. On relatively simple, open-source methods, this method has

been effective, and has improved upon the efficiency of manual metamorphic relation generation. However, these results cannot definitively prove this method will scale to industrial sized software, especially in a system interacting with multiple software artifacts.

In terms of internal validity, there are potential faults in the third party tools used in our study, including scikit-learn, Soot, and QN-S3VM. Scikit-learn and Soot are widely used in the computer science community and are generally accepted to be well-tested and to have a few number of faults. QN-S3VM is a less widely-used tool, and has a higher likelihood of containing more faults that scikit-learn or Soot.

In terms of conclusion validity, there are potential faults in the conclusions drawn from our results. To avoid this, we performed a paired t-test for each set of results to determine if the results are statistically significant. Second, accuracy was used as an evaluation metric. This metric can produce biased results when the data classes are not balanced. Finally, the data points come from different distributions, which affects conclusion validity.

Conclusion and Future Work

We have presented a technique to predict metamorphic relations from Java methods. We built a supervised model using an SVM, and compared it against a semi-supervised model, both using the explicit feature sets extracted from the graph representations of the program under test. Next we built an SVM and compared it to a semi-supervised support vector machine, both using first the random walk kernel, then the graphlet kernel.

We found that label propagation performed better than the SVM for 5 out of the 6 MRs, and the S3VM performed better than the SVM for 4 out of the 6 MRs. These results lead to the conclusion that unlabeled data increases the prediction accuracy

of a binary metamorphic relation prediction classifier.

To expand on this work in the future, we would like to use the label spreading algorithm instead of label propagation. This algorithm allows the α parameter to be relaxed so that labels are not clamped. To improve the three models we built, several steps could be taken in the future. First, more data is essential for improving classification accuracy. This could come in the form of any open source Java methods. The methods used in this study were chosen because they perform actions commonly used in scientific computing, which is where the six studied MRs are usually found. Other types of methods could still add value to the models. Additionally, modifying this work in the future to predict MRs for other languages could be very useful to the scientific community. While Java is a common language in software engineering, there are other more common languages among scientists, including Python, C, and Fortran. Finally, modifying the feature set could improve classification accuracy. It is possible that the features used in our models are too specific. Certain features could be combined, for example, node labels that perform similar actions such as addition and multiplication. Path lengths could also be capped at a set amount, which would also decrease the feature set size. A decreased feature set combined with a increased data set could have a large impact on the accuracy of each model.

REFERENCES CITED

[1] E. Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[2] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2015.

[4] K. P. Bennett and A. Demiriz. Semi-supervised support vector machines. In *Advances in Neural Information processing systems*, pages 368–374, 1999.

[5] N. Cristianini and J. Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[6] F. Gieseke, A. Airola, T. Pahikkala, and O. Kramer. Sparse quasi-newton optimization for semi-supervised support vector machines. In *ICPRAM (1)*, pages 45–54, 2012.

[7] B. Hardin and U. Kanewala. Using semi-supervised learning for predicting metamorphic relations. 2018.

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

[9] U. Kanewala, J. M. Bieman, and A. Ben-Hur. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software Testing, Verification and Reliability*, 26(3):245–269, 2016. stvr.1594.

[10] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[11] H. Liu, X. Liu, and T. Y. Chen. A new method for constructing metamorphic relations. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 59–68. IEEE, 2012.

[12] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, pages 1–4. ACM, 2010.

[13] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. In *SEKE*, volume 8, pages 867–872, 2008.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[15] J. Ratsaby and S. S. Venkatesh. Learning from a mixture of labeled and unlabeled examples with parametric side information. In *COLT*, 1995.

[16] F.-H. Su, J. Bell, C. Murphy, and G. Kaiser. Dynamic inference of likely metamorphic properties to support differential testing. In *Automation of Software Test (AST), 2015 IEEE/ACM 10th International Workshop on*, pages 55–59. IEEE, 2015.

[17] X. Zhu. Semi-supervised learning literature survey. 2005.

[18] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. 2002.