Machine Learning and Event-Based Software Testing: Classifiers for Identifying Infeasible GUI Event Sequences

Robert Gove[1,2], Jorge Faytong[2]
[1]Human-Computer Interaction Lab
[2]Department of Computer Science
University of Maryland
College Park, MD, USA
{rpgove, jfaytong}@cs.umd.edu

September 26, 2011

Abstract

Machine learning is a technique not widely used in software testing even though the broader field of software engineering has used machine learning to solve many problems. In this chapter we present an overview of machine learning approaches for many problems in software testing, including test suite reduction, regression testing, and faulty statement identification. We also present our work using machine learning to augment automatic model-based test case generation for graphical user interfaces (GUIs). A model of the GUI is used to generate test cases, which are event sequences, to be executed on the system under test. Such models may not completely represent the GUI, and therefore may allow infeasible test cases to be generated that violate contraints in the GUI. We use two different machine learning techniques (support vector machines and grammar induction) to identify infeasible test cases (i.e., test requirements that cannot be satisfied). We demonstrate that these techniques are robust across different-length test cases and different GUI constraints.

1 Introduction

Software testing has been widely acknowledged as an important step in the software development process. It also remains a fertile area for applying machine learning techniques

to improve the software testing process. Machine learning is a technique not widely used in software testing [2] even though the broader field of software engineering has used machine learning to solve many problems. One potential area to apply machine learning is to event-based testing in general, and graphical user interface (GUI) testing in particular.

As GUIs become nearly ubiquitous, researchers have proposed several methods for testing GUIs. Some researchers propose methods to help software users avoid bugs in GUIs [23], but other approaches focus on testing the GUI to identify bugs. One such method is to build a directed-graph based model of the GUI [22] that can be used to automatically generate test cases for the GUI. The event-flow model captures all possible event sequences in the GUI; however, some event sequences may be prohibited by state-based constraints [16] (more details of the graph-based models are given in Section 2). Thus the event sequence that comprises a test case may be *infeasible*, meaning that the event sequence contains at least one event that is expected to be available at that point during execution but the event is not allowed by the GUI's state. An event could be unavailable for a number of reasons, possibly because of a bug in the GUI or because of a constraint between events in the GUI specification. This is one example of an opportunity to apply machine learning in the software testing process. Previous work by Huang et al. [16] used a genetic algorithm to repair these infeasible test cases. The authors of this chapter take an alternate approach and use supervised machine learning algorithms to predict which test cases will be infeasible. We will use these algorithms to make predictions by analyzing actual executions of real test cases.

In this chapter we summarize some existing work applying machine learning to software testing, and then we present two methods for predicting event-sequence test case infeasibility: support vector machines (SVMs) [28] and induced grammars [13].

SVMs are a family of supervised learning algorithms for classification and regression analysis. SVMs construct a maximum margin hyperplane to predict which binary label should

be applied; however, SVMs require that input data points (i.e., test cases) be converted to real-valued vectors. In Section 3.1 we discuss four feature extraction techniques to construct real-valued vectors from test cases. Many people consider SVMs to be one of the best off-the-shelf classifiers currently available; for this reason we have applied it to the present problem of classifying test cases.

For grammar induction, our other approach is to induce grammars from the infeasible test cases and use the grammars to classify other test cases. In many cases it is likely that GUI test case generation could be thought of as sentence generation from a grammar G, where infeasible test cases would be generated from a grammar G' whose sentences are a subset of the sentences generated by G. To see this, consider that the sequence of events in a GUI test case must be valid sequences as defined by the event-flow model [21], but an infeasible test case will violate constraints in the GUI. (This assumes that events will always be unavailable because of constraints among events rather than due to external conditions such as a button disabled at certain times of day.) Thus we would like to learn the grammar(s) which generate infeasible test cases. (Note that there may be multiple grammars that need to be learned since there may be multiple constraints in the GUI.)

The two classifiers show two approaches: SVMs are known to be highly effective in many different fields but may not necessarily reveal the causes of infeasible test cases, whereas grammar induction is considered to be a hard problem [25] but the results could potentially yield human-readable results that allow the software tester to identify the causes of infeasible test cases. By predicting which test cases are infeasible, the human software tester may choose a course of action, such as removing the predicted infeasible test cases before they are executed, prioritizing the test suite, or examining the test cases to determine why they are predicted to be infeasible.

We evaluated each test case classification technique on seven subject applications which contain realistic patterns of infeasible event sequences. Overall both classifiers showed

promising results: on average across the subject applications SVMs correctly classified up to 95% of the test cases depending on the feature extraction algorithm and the test case length, and induced grammars correctly classified up to 80% of the test cases depending on the test case length. However, grammar induction performed considerably slower than SVMs which limited the amount of data we were able to collect for grammar induction.

2 Background and Related Work

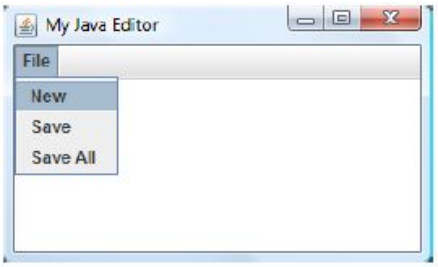
We present the following outline of background material on (1) graph model-based GUI testing, (2) using machine learning techniques for software testing, (3) support vector machines, and (4) grammar induction. We focus our attention on reviewing techniques related to software testing since other authors have reviewed the use of machine learning in the broader field of software engineering, e.g. [31].

2.1 Graph Model-based GUI Testing

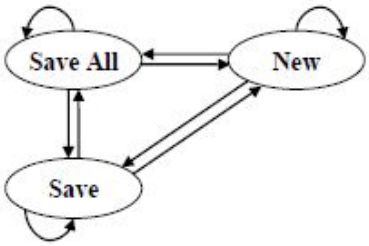
Our work focuses on classifying GUI test cases generated from a graphical model, so we provide a brief discussion of graphical models in GUI testing. Related work on automated model-based GUI testing uses specifications and Finite State Machines to automatically generate test cases [1] [7], but here we restrict our discussion to graphical models.

An *event-interaction graph* (EIG) is a model of a GUI that represents all possible event sequences that can be executed on the GUI. The EIG is a directed graph where each node is a GUI event (such as pressing a button, but not including opening/closing menus or opening new windows) and an edge between two nodes means that one event can be executed immediately after the other. See Figure 1 for an example GUI and its EIG.

After an EIG has been generated for a GUI, test cases can be generated by going node-to-node in the graph to produce sequences of events. However, the EIG generated for a GUI



(a)



(b)

Figure 1: (a) A simple GUI. (b) The EIG for the GUI. Taken from Huang et al. [16].

through GUI Ripping [22] may be only an approximation that does not necessarily capture a complete or correct representation of the GUI. Furthermore, the EIG does not capture state-based relationships. Therefore, event sequences generated from the EIG may contain *infeasible* sequences (test requirements that cannot be satisfied—i.e., there is at least one event that the model expects to be available but is actually unavailable/disabled at that moment during sequence execution). (Memon offers more information about EIGs [21]).

Huang et al. [16] implemented a genetic algorithm to automatically repair infeasible test cases generated from EIGs. However, this approach modifies the underlying test suite, which may be preferred if the software tester needs a feasible test suite, but it also may be undesirable if the test suite has other properties the tester wishes to retain. Furthermore, this approach requires repeatedly executing modified test cases to determine whether they are infeasible, which potentially slows down test suite execution by executing many times

more test cases than were in the original test suite. The constraints that cause a test case to be infeasible may change as the software evolves, faults are fixed, and new faults are introduced.

2.2 Machine Learning and Software Testing

Many research endeavors, like the ones conducted by Hoefel and Elkan [14], have been performed using supervised machine learning algorithms. Several, like Chen et al [6], have used several different classification approaches, of which support vector machines (SVMs) are very popular and are considered to be one of the best off-the-shelf classifiers for supervised learning problems. These classifiers work by processing a set of data and applying a *class* or *label* to each item. A simple example is binary classification, e.g. to label a person as male or female.

One common characteristic of many machine learning techniques is that some data preprocessing must be performed in order to provide the classifier with appropriate features from which to learn. This step is known as *feature extraction*. One feature extraction method of particular interest is pairwise combination generation. It is often implemented for problems involving sequenced strings as input. An example of research implementing this type of transformation can be found in Liao and Noble [20], where pairwise sequence combination similarity and support vector machines are combined for homology detection.

Last describes several potential benefits of using data mining in software testing [18]: Automatically inducing relationships between inputs and outputs, modelling software programs, and producing test suites of non-redundant test cases. However, little work has been performed in the software testing field using machine learning [2]; although Murphy et al. approach the opposite problem and investigate testing machine learning applications, focusing on applications with ranking algorithms [24]. The overall framework relies on manual inspection of the problem domain, the machine learning application, and the runtime options

for the machine learning application.

Several machine learning methods have been proposed for *fault localization*, the process of identifying where a fault lies within the application code. Gondra [12] outlines the use of machine learning for improved fault localization. Renieres and Reiss [26] used the nearest neighbor unsupervised learning algorithm to identify differences in code coverage between a single passing execution and a failing execution. The differences between the executions are likely to contain faulty statements.

Classification trees are also used to locate faults in code [5]. The C4.5 decision tree algorithm is used to identify failure conditions in test executions. Code statements covered by test cases in a particular failure condition are then ranked to identify statements most likely to identify suspicious statements that might contain faults.

Decision trees have also been used by Briand et al. to learn about relationships among conditions in category-partition test suites in order to aid the software tester in understanding the strengths and weaknesses of the test suite [4]. Additionally, Yilmaz et al. [32] use classification trees for testing system configurations. These models are built from a training set of system configurations with known failure/success outcomes.

Neural networks have been another hot topic. They were in two experiments conducted by von Mayrhauser et al. [29]: the first to predict the severity of system failures from test cases, and the second to predict branch coverage from a given test case. For predicting the severity of system failures, the authors chose to use test metrics for the test case encoding. These metrics include test case length, command frequencies, and parameter use frequencies. To predict branch coverage in white box testing, the authors tested a VHDL model and used the model's binary inputs as the inputs to their neural network model. The authors' use of neural networks was successful, but they recommended careful choice of input representation because the neural network used in white box testing had difficulty generating new test cases, which was one of the authors' goals.

Last discusses several uses of data mining for software testing [18]. One of these uses is applying data mining to the task of blackbox testing [17] [19]. The authors use an approach called the info-fuzzy network to determine which input variables are relevant to each output variable. Info-fuzzy networks are similar to decision trees, but are graphs instead of trees. Using the info-fuzzy network, Last and Kandel identified input variables that are relevant to the ouput variables of a software application. Furthermore, their method can be used to identify equivalence classes of test input and produce a set of non-redundant test cases.

But there has also been some work using maching learning in graphical models. Baskiotis et al. [2] developed a system called EXIST that retrieves feasible paths on control flow graphs (CFGs). EXIST relies on an initial labeling of the CFG and the use of a constraint solver. CFGs differ somewhat from EIGs because constraints are built into the CFG. Moreover, ideally there would be a general method to predict infeasible GUI test cases regardless of the underlying model that generated them.

2.3 Support Vector Machines

Support vector machines (SVMs) are a set of related supervised learning methods, which are popular for performing classification and regression analysis using data analysis and pattern recognition. Methods vary on the structure and attributes of the classifier. The most commonly known SVM is a linear classifier, predicting each input's member class between two possible classifications. A more accurate definition would state that a support vector machine builds a hyperplane or set of hyperplanes to classify all inputs in a high dimensional or even infinite space. The closest values to the classification margin are known as support vectors. The SVM's goal is to maximize the margin between the hyperplane and the support vectors.

Support vector machines are very popular and many consider them as the best off-the shelf classifier. Furthermore, there are a wide selection of environments and toolboxes that

implement SVMs. For these reasons we chose to apply SVMs to the problem of classifying infeasible test cases.

2.4 Grammar Induction

It is not immediately obvious which is the best technique for classifying strings of sequential data. Other research has similar goals to ours, but differs in key points: they wish to predict events in the near future [30], work on long sequences [11], learn from noisy data [10], learn a single grammar [3], use machine learning to refine other results [9], learn features from non-sequential data [13], or use algorithms that make assumptions incompatible with event-sequence data [8].

Grammar induction seems to be a promising approach for learning infeasible event sequences, although grammar induction is acknowledged to be a hard problem [25]. The general idea is to learn a grammar $G = (N, \Sigma, P, S)$ from a set of data, where N is the set of nonterminal symbols, Σ is the set of terminal symbols which are the lexical elements of the language, P is the set of production rules which define how sentences are constructed from terminals and nonterminals, and S is the set of start symbols. If the exact grammar is learned, then the language L of the data is represented by $L(G)$.

There are many types of grammars which vary widely in their expressive power. Some of the least expressive grammars are Regular Grammars; yet despite having comparatively little power, Regular Grammars in the form of regular expressions are still an extremely powerful tool for string processing. Researchers have inferred general regular expressions from noisy sequences [10], and induced Regular Grammars from XML documents to infer Document Type Definitions [3].

Another type grammar is the Context-Free Grammar (CFG), which is more expressive than Regular Grammars [15] and could conceivably capture a wide variety of GUI constraints. Some grammar induction techniques induce CFGs [27], but CFGs may be too general for

detecting infeasible test cases. Ideally we would like to learn the simplest possible model that explains our data, and as explained in Section 4.1 the constraints in our data can all be modeled using regular grammars which are a subset of CFGs [15].

3 Methods

In this section we describe our methods for predicting which test cases are feasible and which as infeasible. We describe two methods: the first transforms the test cases into real-valued vectors that are used to train support vector machines to classify test cases, and the second method learns a set of regular expressions which that are used to match infeasible test cases.

3.1 Support Vector Machines

We used the SVM implementation provided in Matlab 7.10.0.499, `svmtrain()` and `svmclassify()`, using a Gaussian Radial Basis Function kernel with a default scaling factor of 1. This method offers great classification flexibility in that it is able to set more complex margins than linear classifiers while not losing the kernel invariability in space characteristic, from which polynomial classifiers suffer.

We applied SVM classifiers following standard procedures for all machine learning techniques. It is commonly known that machine learning techniques include a feature extraction phase, so that the data is shaped in a way that the algorithm can work with and consequently obtain the best possible results.

SVMs require that the data be real-valued vectors; however, test cases are sequences of IDs which need to be converted to real-valued vectors. In order to identify useful feature extraction algorithms to create the vectors, we implemented algorithms to generate four potential SVM input vectors that we call Basic, Pairwise, Full Pairwise, and Reduced Pairwise. We then compared the classification accuracy for each type of input vector, which we discuss

in Section 5.3.

Having this in mind, the generation of the Basic, Pairwise, Full Pairwise, and Reduced Pairwise vectors consisted of four stages.

The first stage handles the value assignment of the first n vector attributes, where n is the number of different IDs in the GUI. Index i in the resulting vector corresponds to the number of times that event i appears in the test case. The output of this stage is the n-dimensional Basic vector, but this output is also used for Pairwise and Full Pairwise vectors. For example, if we have a GUI with event IDs in $\{0, 1, 2\}$ and we have the original test case

```
0 1 2 1 2
```

then the Basic vector would be

```
1 2 2
```

In this step, we simply count the number of appearances of each ID.

The second stage is an intermediate step which determines all the possible pairwise combinations. In other words, we create a vector P which lists all the possible combinations of the available IDs by iterating from the lowest ID to the highest. We exclude combinations of the same ID, i.e.,

```
22
```

Each event ID i will have $n - 1$ ordered pairs that start with ID i, where n is the number of different IDs in the GUI. For the input vector above, the resulting combinations vector P would be:

```
01 02 10 12 20 21
```

The third stage then iterates through the input test case counting all the appearances of each combination found in vector P (e.g., counting the number of times that `0` occurs

immediately before 1, the number of times 0 occurs immediately before 2, etc.). Each count is appended to the Basic vector, so that the n^2-dimensional Pairwise vector looks like:

```
1 2 2 1 0 0 2 0 1
```

where 1 2 2 is the original Basic vector and

```
1 0 0 2 0 1
```

are the counts of each pair in P that occurs in the test case.

The fourth stage iterates through the test case counting all the appearances of each generated combination as well, with one difference to the third stage: this run counts all the times the second pair ID appears after the first pair ID in the given test case. For example, for the pair 01, this stage counts the number of times 1 occurs anywhere after 0 in the test case. Likewise, this process is performed for every pair. Each of these results is appended to the Basic vector as well, so that the final N^2-dimensional Full Pairwise vector for the above test case would look like:

```
1 2 2 2 2 0 2 0 1
```

We also created a fourth feature vector called Reduced Pairwise that is similar to Pairwise. The difference is that instead of counting the frequencies for all $n^2 - n$ pairs, it only counts frequencies for the n most frequently occuring pairs in the test suite. As with Pairwise, the frequencies for the n most occuring pairs are appended to the Basic feature vector (i.e., the frequencies for each of the n events). This results in a feature vector of length $2n$.

Basic, Pairwise, Full Pairwise, and Reduced Pairwise feature vectors were generated for all test cases.

In our data, GUI event IDs may be in $\{0, 1, 2\}$, $\{0, 1, 2, 3\}$, or $\{0, 1, 2, 3, 4\}$ depending on the program analyzed (see Section 4.1). Thus, our transformed vectors will have 9, 16, or 25 attributes respectively. Because the length of the vectors will vary depending on the number

of distinct GUI event IDs, we will need three separate SVM implementations for each vector size.

It is worth mentioning that none of the Basic, Pairwise, Full Pairwise, or Reduced Pairwise vectors can accurately model consecutive events with the same ID, such as the sequence

```
1 1 1 1 1
```

. Although in some GUIs it is possible that repeated events result in an infeasible sequence, repeating an action in a test case for our subject GUIs will not cause the test case to become infeasible. A more general feature vector would account for consecutive events with the same ID.

These feature extraction techniques can be applied to the classification problem. We can apply one of the above feature extraction algorithms (Basic, Pairwise, Full Pairwise, or Reduced Pairwise) to transform each test case in a training test suite T_r (the training data set) into a real-valued vector. This set of real-valued vectors are used by the `svmtrain()` function in Matlab to train a support vector machine (SVM). (As mentioned above, we use a Gaussian Radial Basis Function kernel with a default scaling factor of 1.) After the SVM has been trained, we apply the same feature extraction algorithm to each test case in a second test suite T_t (the testing data set), which once again generates a set of real-valued vectors (one for each test case). Using the `svmclassify()` function in Matlab, we can use the SVM trained in the previous step to classify each test case in T_t as feasible or infeasible. Note that these real-valued vectors can also be used as input to other classification algorithms, but in our approach we use the SVM described above.

3.2 Grammar Induction

In our present application, the desired process is to estimate the conditional probability that a test case is infeasible given that it contains a certain event sequence (string of events). We

assume that infeasible events in a test case are completely determined by the events that precede them, i.e., there is some grammar $G = (N, \Sigma, P, S)$—where Σ is the set of possible events in the GUI, and P characterizes the constraints in the GUI—such that the language of that grammar, $L(G)$, is the set of infeasible test cases on the GUI. Trivially, the set of all test cases is given by $T = L(\Sigma)$. Therefore, the set of feasible test cases is given by $L(\Sigma) \setminus L(G) = \neg L(G)$. These assumptions hold in our subject applications; however, this may not be the case for all real-world applications.

It follows that each test case t is a sentence which can be generated by G. Therefore, we would like to learn a set of rules so that we can correctly classify, for all test cases t in a test suite T, whether $t \in L(G)$. Clearly, N, Σ, and P will vary depending on the GUI and its underlying application, so a separate set of grammars will need to be induced from each subject. Next we present a high-level description of the training algorithm, followed by a more detailed psuedocode algorithm.

We will learn a set of regular grammars $R_i = \{r_1, r_2, \ldots, r_k\}$ from each infeasible test case t_i (with $1 \leq i \leq |T| = n$), where each of r_1 is a regular grammar that has a high chance of matching only infeasible test cases. Presently, if a test case matches any rule $r_j \in G = R_1 \cup R_2 \cup \cdots \cup R_n$ then we classify the test case as infeasible; otherwise the test case is marked feasible.

The algorithm starts by using the test case as a regular expression, and then iteratively modifying each regular expression by replacing specific events with general patterns. When modifications no longer improve the quality of the regular expression, it is added to the final set of regular expressions.

Where the objective function $f(r)$ is given by

$$f(r) = \frac{|\{t : r \text{ matches } t \wedge t \text{ is infeasible}\}|^2}{|\{t : t \text{ is infeasible}\}| \cdot |\{t : r \text{ matches } t\}|}. \tag{1}$$

```
1: R ← ∅
2: for all infeasible test cases t ∈ T do
3:   l ← index of last event executed in t
4:   new_regexs ← ∅
5:   new_regexs.push(subsequence of t from 0 to l)
6:   while new_regexs ≠ ∅ do
7:     r ← new_regexs.pop()
8:     for all event indices i ∈ r do
9:       if i is not modified then
10:        for all event e ∈ E do
11:          q ← copy_of(r)
12:          if e ≠ q.index(i) then
13:            q.index(i) ← (# − e)*
14:          else
15:            q.index(i) ← (e)*
16:          end if
17:          if f(q) ≥ f(r) then
18:            new_regexs.push(q)
19:          end if
20:        end for
21:        if no new rules added to new_regexs then
22:          R.push(r)
23:        end if
24:      end if
25:    end for
26:  end while
27: end for
```

Figure 2: Algorithm to generate a set of regular expressions (regexs) that identify infeasible test cases from the training test suite T from events $e \in E$. $f(r)$ is defined in Equation 1

Thus, we are essentially estimating the probability

$$p(r \text{ matches } t | t \text{ is infeasible}) \cdot p(t \text{ is infeasible} | r \text{ matches } t).$$

Selecting regular expressions with a high objective value will increase the likelihood that the regular expression will match as many infeasible test cases as possible and as few feasible test cases as possible. The above optimization technique is a simple hill-climbing algorithm

to find a local maximum.

Although this technique searches for regular grammars that have a high probability of matching infeasible test cases, it does not verify that the constraints in the grammars cause test cases to be infeasible.

We implemented the algorithm in Figure 2 as a Java 6 application. The application uses the algorithm to create a set of regular expressions from a training test suite T_r (the training data set). Then the application uses the regular expressions to classify the test cases in a second test suite T_t (the testing data set): if any of the regular expressions matches a test case in T_t, that test case is marked infeasible; otherwise the test case is assumed to be feasible.

4 Evaluation

Below we describe our test case data and evaluation methodology.

4.1 Test Case Data

We use the UNL.Toy.2010 data from the Comet group, which is a joint effort between the E2 laboratory at UNL and the GUITAR group at UMD[1]. This is the same data used by Huang et al. for GUI test case repair [16].

The data is a set of test suites, where each test suite is a set of test cases. Each test case of length k is composed of a string of k integer tokens which denote GUI events. These are followed by a boolean token indicating whether the test case is feasible, and a 0-based index denoting the failure point (if the test case is feasible then the index is k). For example,

```
1 0 3 2 4 F 3
```

[1] `http://comet.unl.edu/`

is a test case of length 5 which is infeasible and failed on the event at index 3 (which corresponds to event 2). All the test cases in this data are of length 5, 10, 15, or 20. Test suites are comprised of between 55 and 1303 test cases depending on the covering array used to generate the test suite and the number of events in the GUI application. GUI event IDs may be in $\{0, 1, 2\}$, $\{0, 1, 2, 3\}$, or $\{0, 1, 2, 3, 4\}$ depending on the GUI application.

The test cases were generated from six different GUI applications:

- *2cons*: Clicking the Event 1 button disables the Event 2 button, and clicking any button other than Event 1 re-enables Event 2.
- *3cons*: The same as 2cons, except Event 3 is disabled by clicking Event 1 followed immediately by Event 2, and clicking any other button re-enables Event 3.
- *2excl*: Clicking the Event 1 button causes Event 2 to be disabled for the remainder of the program execution.
- *3excl*: The same as 2excl, except Event 3 is disabled by clicking Event 1 followed immediately by Event 2, and Event 3 remains disabled for the remainder of the program execution.
- *disb*: Event 1 is always disabled.
- *reqs*: Event 3 begins disabled, but clicking Event 2 causes Event 3 to be enabled for the remainder of the program execution.
- *cmpd*: This features the 2cons, 3cons, and reqs constraints all in one GUI.

All of these constraints can be modeled with regular grammars. For example, feasible test cases for 2cons would be given by the grammar $((\# - Event1)^*(Event1)^*(\# - Event2)^*)^*$.

The test cases for each of these applications were generated using covering array generation using a combinatorial interaction technique [33]. Each subject application is described

by two variables: one for the CIT model and one for the explicit constraints in the application.

The covering array file has the following format:

- The strength of testing (all t-way combinations of events).
- The test case length (k events in the test case).

For example, if we are creating a 2-way test case of length $k = 5$, the covering array would be abbreviated as *t2k5*. See Huang et al. [16] for a full description of the applications and the test case generation methodology.

4.2 Partitioning the Test Case Data

In order for the classification algorithms to learn to classify feasible and infeasible test cases, the training data must include both types of test cases. To ensure the training data includes both feasible and infeasible test cases, we partition the data as follows.

For a test suite T, we create five pairs (T_r, T_t) by breaking T in two parts, where T_r is the training data set and T_t is the testing data set. For each pair, we take $x\%$ of the test cases in T and put them in T_r, and the remaining $(100 - x)\%$ of the test cases are put in T_t. To generate each of the five pairs, we repeat this procedure for $x \in \{10, 20, 30, 40, 50\}$.

To generate T_r and T_t from test cases t in test suite T, our application performs the following procedure:

1. Create the sets $I = \{t \in T | t \text{ is infeasible}\}$ and $F = \{t \in T | t \text{ is feasible}\}$.
2. If $|I| = 1$ (or $|F| = 1$) then add the test case in I (or F) to the set of training test cases T_r, and add $x\%$ of the test cases in F (or I) to T_r and add the remaining test cases to T_t; otherwise add $x\%$ of the I and $x\%$ of F to T_r and put the remaining test cases in T_t. In otherwords, put $x\%$ of the feasible test cases and $x\%$ of the infeasible

test cases in T_r and the remaining test cases in T_t, and ensure that T_r contains at least one feasible test case and at least one infeasible test case.

Our application does not attempt to randomize the assignment of test cases to the training or testing data sets.

The decision to include at least one test case of each type (if possible) is justified because when a software tester is executing a test suite, there is no need to prioritize test cases unless it is known that infeasible test cases exist, and this can only be known at execution time. As long as I is empty there is no need to run a classifier.

4.3 Implementation

The overall implementation consisted of several parts, which were described in detail in previous sections:

- *Partitioner*: Partition the test suite T into T_r and T_t.
- *SVM Extractor*: Apply the Basic, Pairwise, Full Pairwise, or Reduced Pairwise feature extraction algorithm to each test case in T and create a real-valued vector for each test case.
- *SVM Trainer*: Use the `svmtrain()` function in Matlab to train a radial basis function SVM classifier with the real-valued vectors from T_r generated in the *SVM Extractor* step.
- *SVM Classifier*: Use the `svmclassify()` function in Matlab to classify the T_t real-valued vectors generated in the *SVM Extractor* step using the SVM trained in the *SVM Trainer* step.
- *RegEx Inducer*: Create regular expressions from the test cases in T_r using the grammar induction algorithm in Figure 1.

- *RegEx Classifier*: Use the induced regular expressions from the *RegEx Inducer* step to classify test cases in T_t (if any regular expression matches a test case in T_t, the test case is classified infeasible; otherwise it is classified as feasible).

Note that the SVM steps can be run separately from the RegEx steps, but all steps rely on partitioning the test suite into the training set and the test set. An implementation in a production system might skip the *Partitioner* step and simply run the *SVM Extractor* and *SVM Trainer* steps (or the *RegEx Inducer* and *RegEx Classifier* steps) on a percentage of random test cases in the test suite that have already been classified as feasible or infeasible. Then this output could be used in the classifier step to classify the remainder of the test cases, or classify a newly generated set of test cases.

4.4 Experimental Design

We evaluated our algorithms by addressing the following questions:

- **Question 1**: What is the effect of training data set size on classification errors?
- **Question 2**: Can a classifier trained on one test case length correctly classify test cases of a different length?

It is worth mentioning that application 3excl does not have t2k5, t2k10, t2k15, and t2k20 covering arrays. Therefore, when computing the average results, only t3k5 and t3k10 include 3excl.

4.5 Question 1

To answer Q1, we trained the SVM classifiers on a percentage of the test cases for each cover array and each GUI application, and then used the remaining test cases to test the classification error. The percent size of the training set was 10%, 20%, 30%, 40%, and 50%

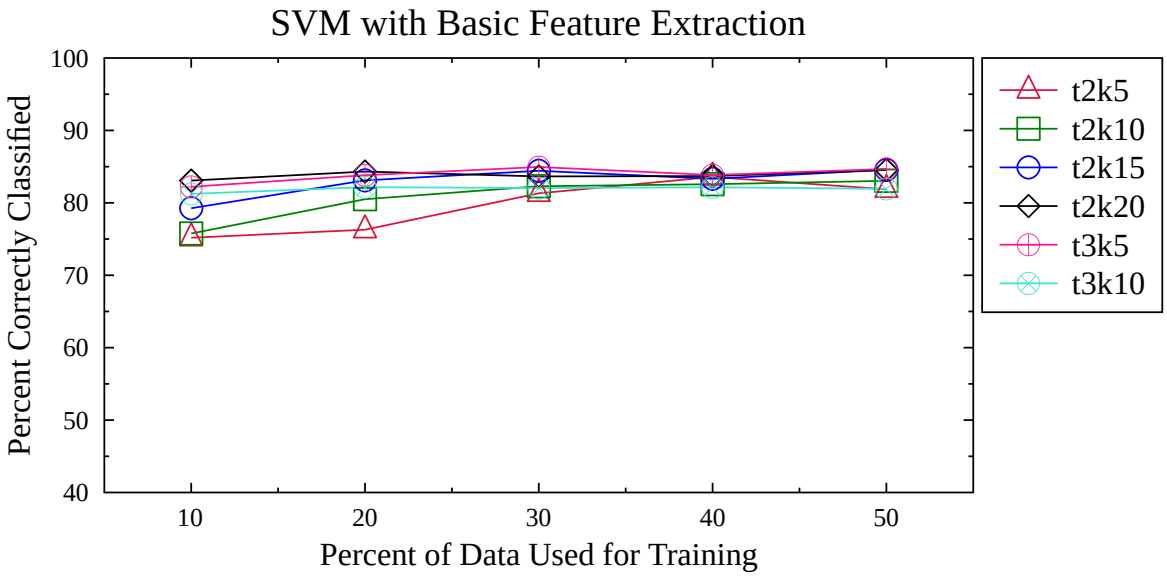


Figure 3: The plot shows the results for the average results of the Basic algorithm's performance.

of the test cases for that GUI application and covering array. We then averaged the percent correct classification across all GUI applications. This data is shown in Figures 3, 4, 5, and 6.

We attempted to test the effect of training data set size on the effectiveness of the induced grammars; however, the training algorithm was too slow in all cases except when training on 10% of the data. Moreover, we did not have sufficient time to allow the algorithm to finish training on test cases longer than length 10. Figure 7 shows the average results across all GUI applications for training and classifying t2k5 and t2k10.

4.6 Question 2

To answer Q2, we trained the SVMs on one data set with test cases of a certain length and then tested the classifier on test cases of different lengths. The classification rates are shown in Figures 8, 9, 10, and 11. In all cases the SVMs used 30% of the data during the training

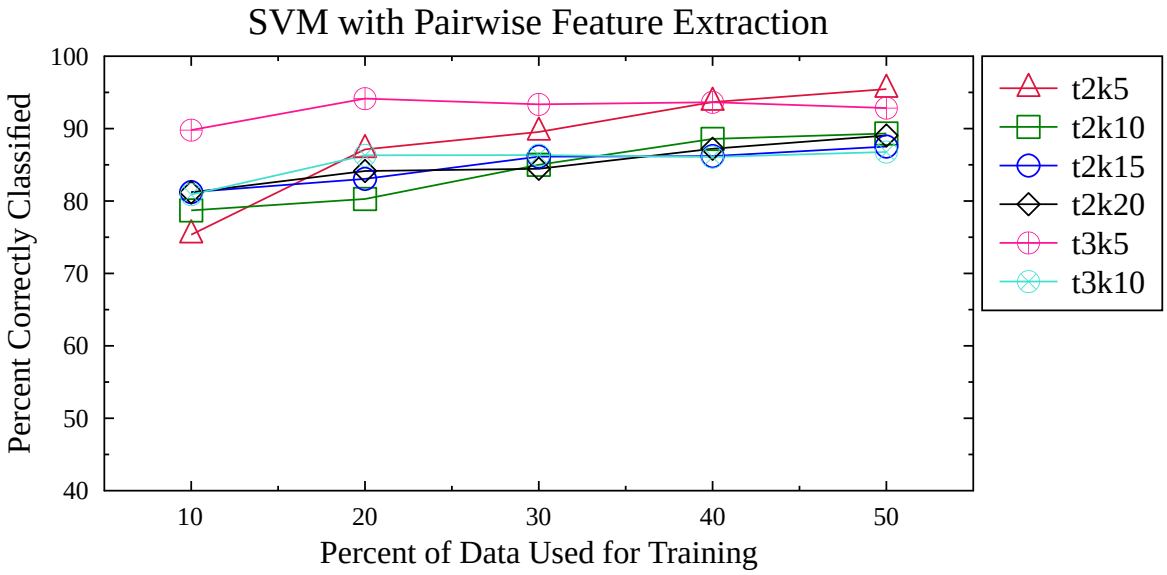


Figure 4: The plot shows the results for the average results of the Pairwise algorithm's performance.

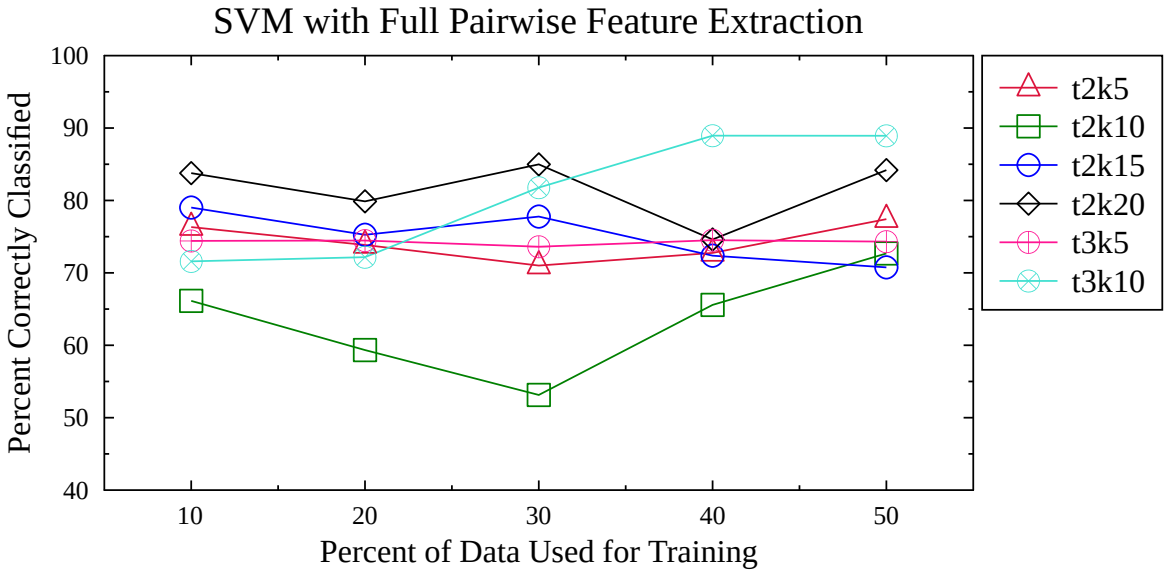


Figure 5: The plot shows the results for the average results of the Full Pairwise algorithm's performance.

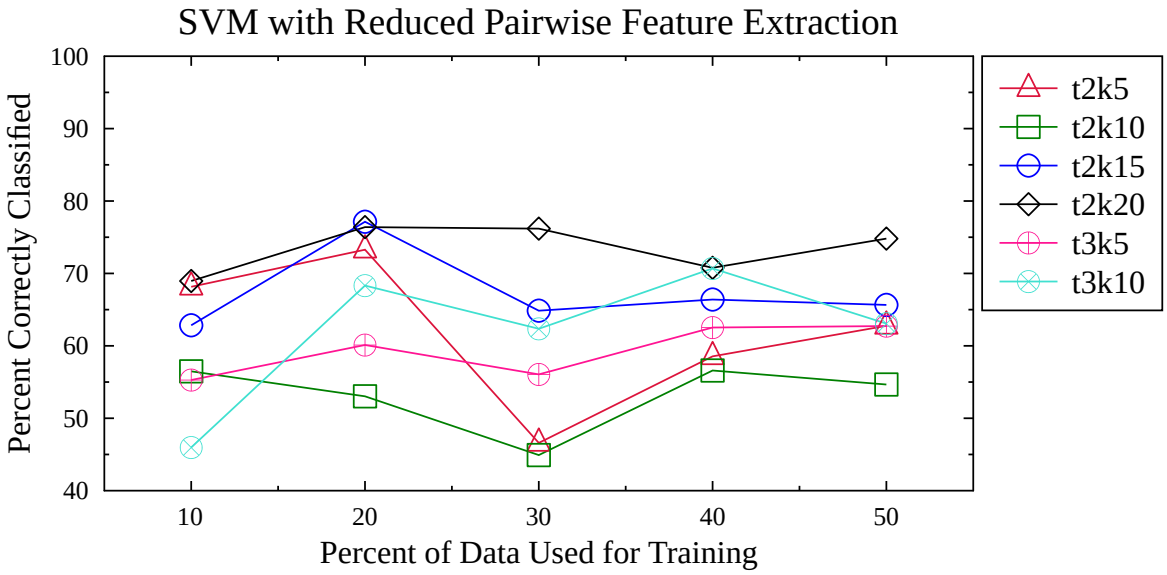


Figure 6: The plot shows the results for the average results of the Reduced Pairwise algorithm's performance.

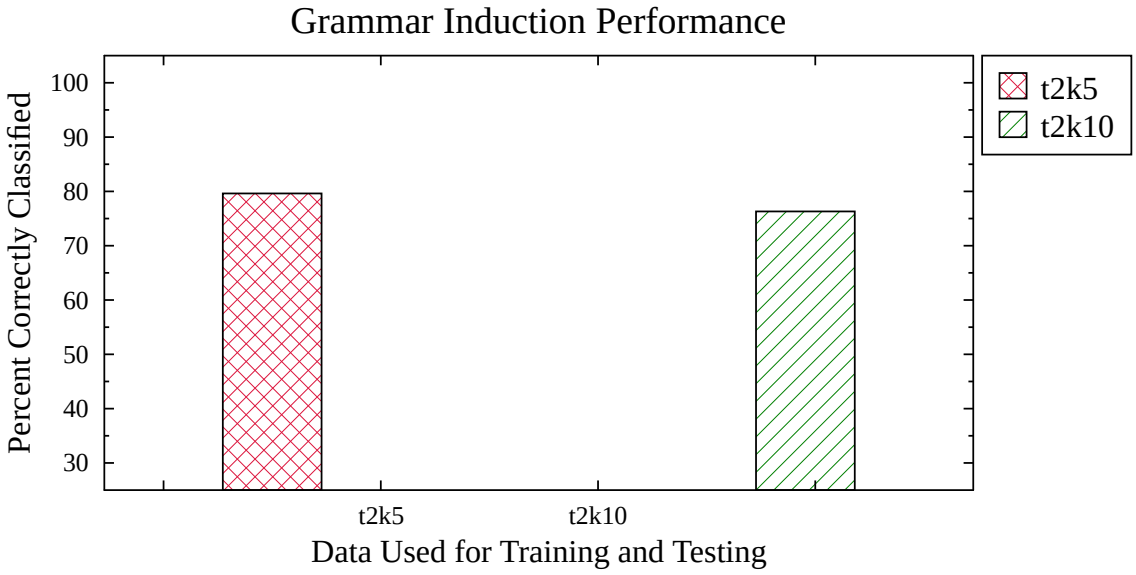


Figure 7: The average correct classification rate across all GUIs for induced grammars. The size of the training data is fixed at 10% of the data.

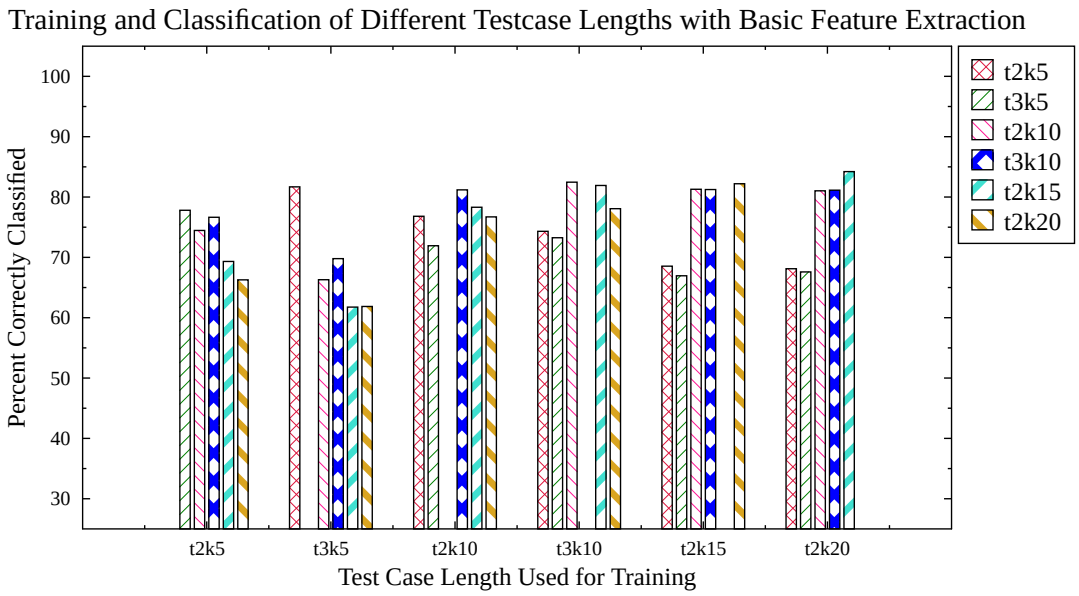


Figure 8: A graph of the average performance of the Basic algorithm for each covering array as training data vs. the remaining arrays as test data.

phase and the remaining 70% during the testing phase, based on the results from Question 1.

We performed a similar experiment with grammar induction, but using 10% of the data for training and 90% for the testing phase. These results are shown in Figure 12.

5 Results

5.1 Question 1

From Figures 3 and 4 we see that there is a moderate improvement in classification rate as the SVMs with Basic and Pairwise feature extractions train on larger data sets, but only the SVM with Pairwise shows any continued improvement when training on more than 30% of the data sets. In Figure 5 we see the Full Pairwise SVM has a large variation in

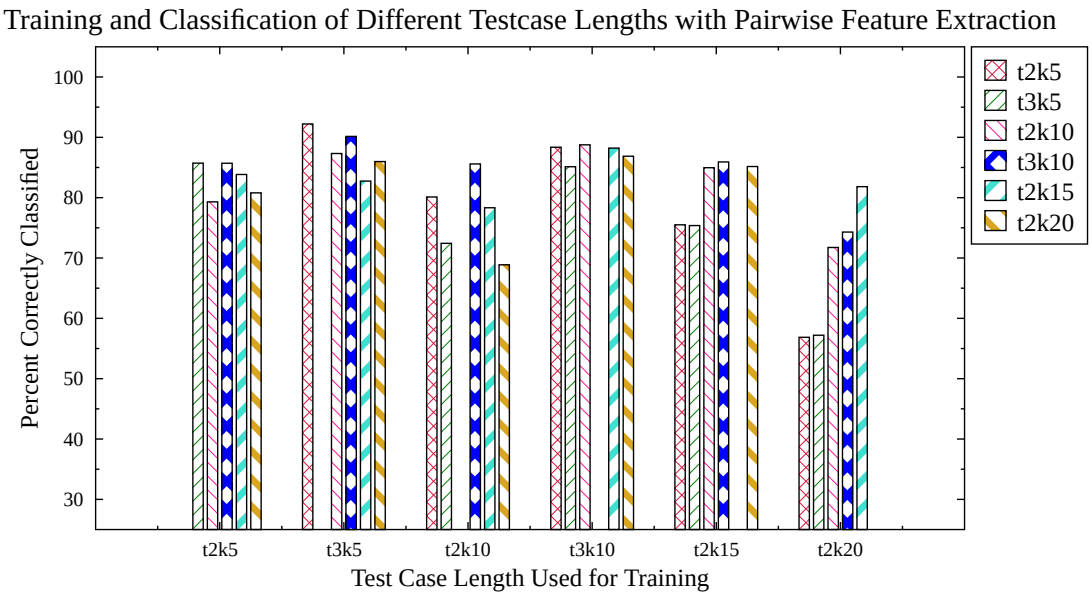


Figure 9: A graph of the average performance of the Pairwise algorithm for each covering array as training data vs. the remaining arrays as test data.

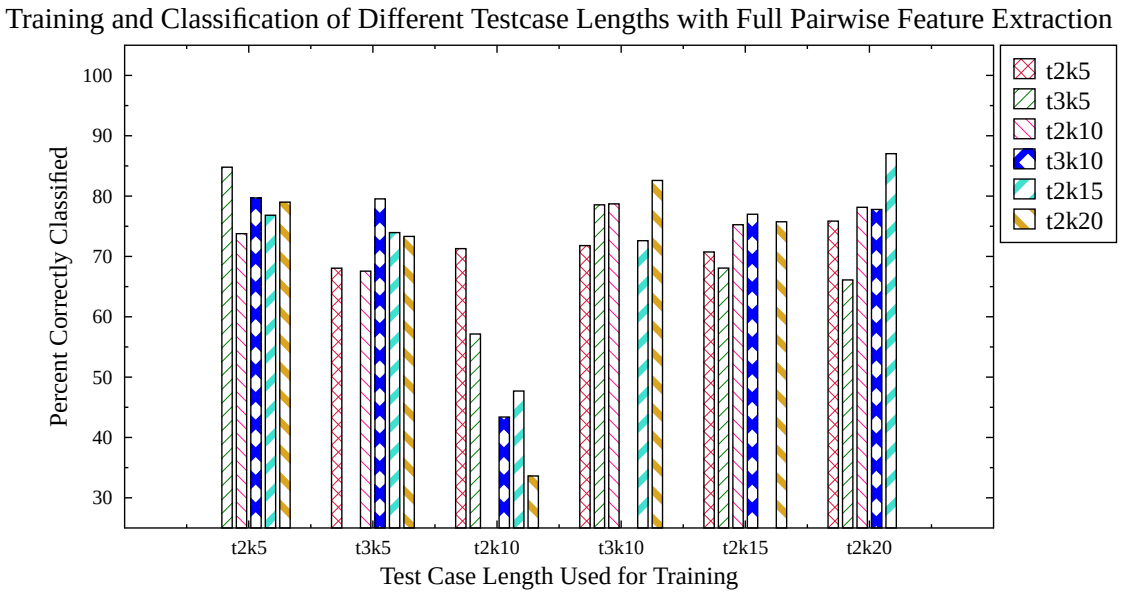


Figure 10: A graph of the average performance of the Full Pairwise algorithm for each covering array as training data vs. the remaining arrays as test data.

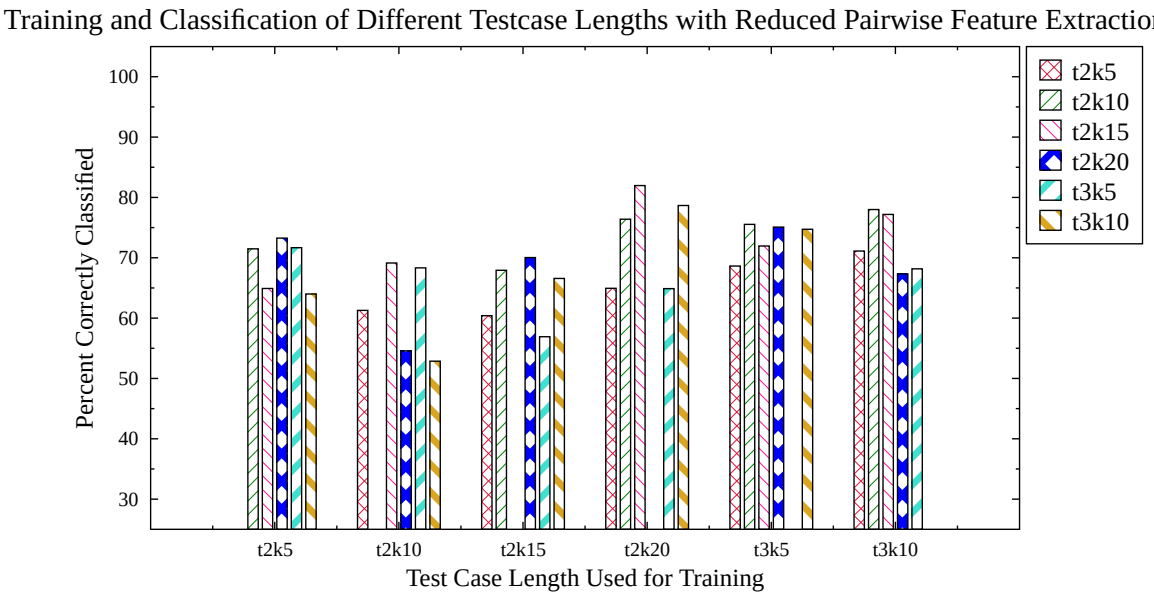


Figure 11: A graph of the average performance of the Reduced Pairwise algorithm for each covering array as training data vs. the remaining arrays as test data.

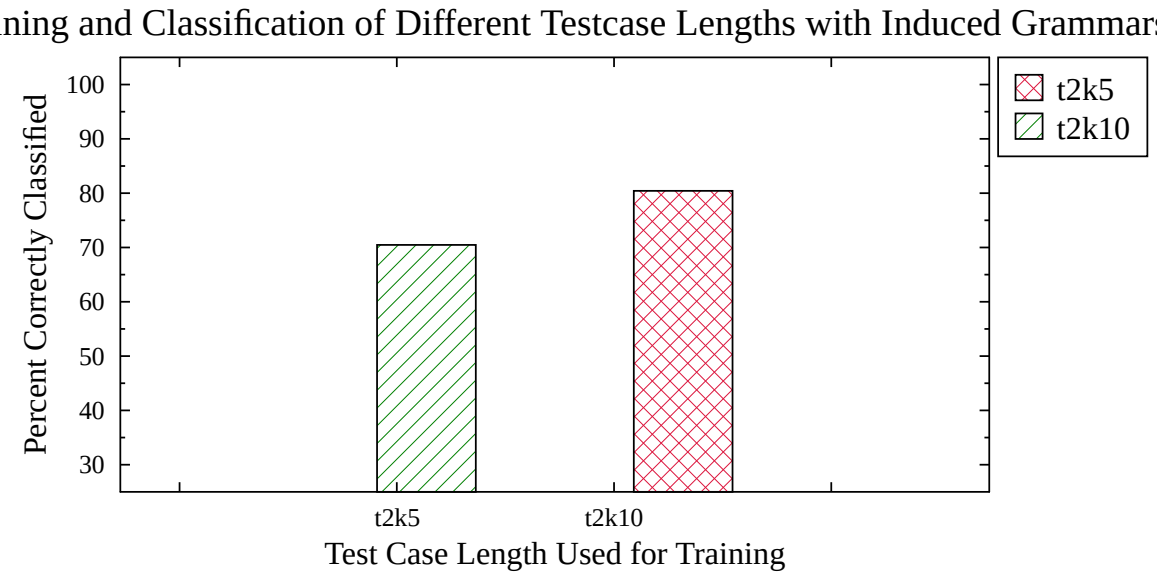


Figure 12: A graph of the average performance of the induced grammars for two covering arrays as training data vs. the remaining array as test data.

its performance and has no clear pattern as we change the size of the training data set. Similarly, Figure 6 shows the results for the Reduced Pairwise algorithm, which does not appear to give reliable classification results. With the exception of Reduced Pairwise, these feature extraction techniques we mostly yield correct classification rates at or above 75%.

The data collected for the induced grammars is limited, but from Figure 7 we see that even when training on only 10% of the data it correctly classifies the data close to 80% of the time.

From these results we concluded that we should use 30% of the data during the training phase for the SVMs, and 10% of the data during the training phase for inducing grammars. This answered Question 1.

5.2 Question 2

For the SVMs, we notice a general trend where the classifiers perform best when they are training and testing on data of similar lengths (with the exception of Reduced Pairwise, which does not appear to perform well). Due to lack of data we are unable to draw general conclusions for the induced grammars, but we can say that for the data shown in Figure 12 the grammar induction performs above 70% correctly in both cases, but the difference between the two sets of trained grammars is 10%.

This answered Question 2, and shows that the SVM and feature extraction methods are moderately robust to changes in test case length, despite the fact that the values in the feature vectors are highly sensitive to the test case length.

5.3 Discussion

We have observed encouraging results for Question 1. This demonstrates the effectiveness of machine learning techniques in general, and SVMs and grammar induction in particular,

for predicting test case feasibility. As stated previously, determining infeasible test cases is difficult for known automated software testing tools. This work offers the community an innovative and effective method to help solve this complex problem.

The classification rates shown in this chapter are an average across several GUI applications with different event constraints. The classification results remain high despite this variation, which demonstrates a robustness across GUI applications.

SVMs using Full Pairwise feature vectors demonstrated results that are more chaotic than the Basic and Pairwise feature vectors, but despite this the classification results were still high in many cases. Overall, however, simple feature extraction techniques may be equally as effective as more complex techniques while providing more consistent results.

Some test suites have a very high density of infeasible test cases; in some cases close to or equal to 100%. This is likely caused by the simple nature of the GUI applications, which have between three and five different events. This greatly increases the likelihood that longer test cases will attempt to execute an infeasible event sequence. These two features could potentially affect the validity of our results since this may not be indicative of real-world GUI applications. Furthermore, if a test suite contains only infeasible test cases then the classifier should trivially correctly classify all test cases in the test suite, which would cause the classifier to perform better on average than it would in a more realistic scenerio.

The classifiers yielded unexpected results when training on test cases of a different length than they were classifying. At first glance, it seems unlikely that SVMs would be able to perform well under these circumstances given that the output of the feature extraction algorithms is highly dependent on the test case length. The results demonstrate that SVMs with Pairwise feature extraction are robust to small changes in test case length between the training set and the testing set, but large differences may result in unsatisfactory results.

There are some limitations to these methods. First, as mentioned in Section 3.1, the feature extraction algorithms do not model consecutive events with the same ID. This may

affect the robustness of the SVM approach if the subject GUI has a constraint where one button may not be pressed twice consecutively (e.g., pressing "Save" and then "Save" again).

Second, the grammar induction algorithm searches for regular expressions that have a high correlation with infeasible test cases. Because regular expressions are an implementation of regular grammars, the learned regular expressions cannot express complex constraints such those modeled by context-free grammars. Furthermore, the regular expressions merely correlate with infeasible test cases; they are not demonstrated to cause infeasibility.

Third, the grammar induction algorithm cannot learn generalized regular expressions such as $(e^*(\# - e)^*)^+$. However, the algorithm in Figure 2 can learn $e^*(\# - e)^*$, and if a test case is matched by the regular expression $(e^*(\# - e)^*)^+$ then it will also be matched by $e^*(\# - e)^*$.

Fourth, SVMs that used Reduced Pairwise feature vectors probably did not perform as expected because our implementation looks for frequent event pairs among all testcases instead of pairs that may be more common to infeasible test cases. Because of this, the SVM is merely classifying on a smaller number of features that are common to both feasible and infeasible testcases.

Fifth, our data sets were generated from relatively simple applications. Although convenient for this initial research, future work should investigate the scalability of these techniques with more complex GUIs. In particular, it would be important to consider subject GUI applications that have the same density of infeasible event sequences as we see in real applications.

6 Conclusion

In this chapter we have reviewed some techniques for applying machine learning to software testing, and we have demonstrated the effectiveness of using two supervised learning algo-

rithms for classifying infeasible test cases: support vector machines, and induced grammars. The results of the induced grammars are limited due to the computational costs, but they demonstrate the effectiveness of this approach. One advantage to grammar induction is that the induced grammars can show software testers the types of event sequences that cause infeasible test cases. An optimized grammar induction algorithm could provide overall good classification results while enabling the software tester to learn about the constraints in the GUI. Furthermore, the grammar induction algorithm described here is very limited in the types of constraints it can discover in the data. Future work could provide a faster algorithm that can induce more complex grammars.

Support vector machines were very effective depending on the feature extraction algorithm. The Pairwise algorithm performed the best for our subject applications, and even demonstrated robustness when training test cases of one length and classifying on test cases of a different length.

The results have demonstrated that classifying test case feasibility is possible. The behavior of the application under test and the consequent understanding of its nature could be proven very useful when selecting the appropriate classification tools. Investigation in this topic would deliver results that save time and resources, thereby extending the availability of resources for testing.

Acknowledgment

The authors wish to thank Atif Memon and Lise Getoor for comments which improved the quality of this work.

References

[1] Nikolai Tillmann Ana Paiva, João C. P. Faria and Raul F. A. M. Vidal. A model-to-implementation mapping tool for automated model-based gui testing. In *Int. Conf. Formal Engineering Methods*, pages 450–464, 2005.

[2] Nicolas Baskiotis, Michèle Sebag, Marie-Claude Gaudel, and Sandrine Gouraud. A machine learning approach for statistical software testing. In *Proc. 20th Int. Joint Conf. Artifical Intelligence*, pages 2274–2279, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[3] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 35(2):1–47, 2010.

[4] Lionel C. Briand, Yvan Labiche, Zaheer Bawar, and Nadia Traldi Spido. Using machine learning to refine category-partition test specifications and test suites. *Information and Software Technology*, 51(11):1551 – 1564, 2009. Third IEEE International Workshop on Automation of Software Test (AST 2008); Eighth International Conference on Quality Software (QSIC 2008).

[5] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 137–146, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Huiling Chen, Huan-Xiang Zhou, Xiaohua Hu, and Illhoi Yoo. Classification comparison of prediction of solvent accessibility from protein sequences. In *Proc. 2nd Conf. Asia-Pacific Bioinformatics*, pages 333–338, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[7] Woei-Kae Chen and Zheng-Wen Shen. Gui test-case generation with macro-event contracts. In *Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on*, pages 145 –151, june 2010.

[8] Thomas G. Dietterich. Machine learning for sequential data: A review. In *Proc. Joint IAPR Int. Workshop Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, London, UK, 2002. Springer-Verlag.

[9] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. Design pattern mining enhanced by machine learning. In *Proc. 21st IEEE Int. Conf. Software Maintenance*, pages 295–304, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Ugo Galassi and Attilio Giordana. Learning regular expressions from noisy sequences. In *Proc. 6th Int. Symp. Abstraction, Reformulation and Approximation*, pages 92–106, 2005.

[11] Attilio Giordana, Paolo Terenziani, and Marco Botta. Recognizing and discovering complex events in sequences. In *Proc. 13th Int. Symp. Foundations of Intelligent Systems*, pages 374–382, London, UK, 2002. Springer-Verlag.

[12] Iker Gondra. Applying machine learning to software fault-proneness prediction. *J. Syst. Softw.*, 81(2):186–195, 2008.

[13] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining — a general survey and comparison. *SIGKDD Explor. Newsl.*, 2(1):58–64, 2000.

[14] Guilherme Hoefel and Charles Elkan. Learning a two-stage SVM/CRF sequence classifier. In *Proc. 17th ACM Conf. Information and Knowledge Management*, pages 271–278, New York, NY, USA, 2008. ACM.

[15] JE Hopcroft and JD Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[16] Si Huang, Myra Cohen, and Atif M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proc. 3rd IEEE Int. Conf. Software Testing, Verification and Validation*, Washington, DC, USA, 2010. IEEE Computer Society.

[17] M. Last and A. Kandel. Automated test reduction using an info-fuzzy network. In *Software Engineering with Computational Intelligence*, pages 235–258. Kluwer Academic Publishers, 2003.

[18] Mark Last. Data mining for software testing. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 1239–1248. Springer US, 2005.

[19] Mark Last, Menahem Friedman, and Abraham Kandel. The data mining approach to automated software testing. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 388–396, New York, NY, USA, 2003. ACM.

[20] Li Liao and William Stafford Noble. Combining pairwise sequence similarity and support vector machines for remote protein homology detection. In *Proc. 6th Annu. Int. Conf. Computational Biology*, pages 225–232, New York, NY, USA, 2002. ACM.

[21] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[22] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

[23] Amir Michail and Tao Xie. Helping users avoid bugs in GUI applications. In *Proc. 27th International Conference on Software Engineering (ICSE 2005)*, pages 107–116, May 2005.

[24] C. Murphy, G. Kaiser, and M. Arias. An approach to software testing of machine learning applications. In *Proc of the 19th International Conference on Software Engineering and Knowledge Engineering*, pages 167–172, 2007.

[25] Sara Porat and Jerome A. Feldman. Learning automata from ordered examples. *Machine Learning*, 7:109–138, 1991. 10.1023/A:1022642911489.

[26] M. Renieres and S.P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30 – 39, oct. 2003.

[27] Olgierd Unold. Grammar-based classifier system: a universal tool for grammatical inference. *W. Trans. on Comp.*, 7:1584–1593, October 2008.

[28] Vladimir N. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[29] Anneliese von Mayrhauser, Charles W. Anderson, Tom Chen, Richard Mraz, and C.A. Gideon. On the promise of neural networks to support software testing. In *Computational intelligence in software engineering.* World Scientific Publishing Co., Inc., 1998.

[30] Gary M. Weiss. Timeweaver: a genetic algorithm for identifying predictive patterns in sequences of events. In *Proc. Genetic and Evolutionary Computation Conf.*, pages 718–725. Morgan Kaufmann, 1999.

[31] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. Data mining for software engineering. *IEEE Computer*, 42(8):35–42, August 2009.

[32] C. Yilmaz, M.B. Cohen, and A.A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20 – 34, 2006.

[33] Xun Yuan, Myra Cohen, and Atif M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *Proc. 22nd IEEE Int. Conf. Automated Software Engineering*, Washington, DC, USA, 2007. IEEE Computer Society.