Identifying Infeasible GUI Test Cases Using Support Vector Machines and Induced Grammars

Robert Gove[1,2,3], Jorge Faytong[3]
[1]*Human-Computer Interaction Lab,* [2]*Institute for Advanced Computer Studies*
[3]*Department of Computer Science*
University of Maryland
College Park, MD, USA
{rpgove, jfaytong}@cs.umd.edu

***Abstract*—Model-based GUI software testing is an emerging paradigm for automatically generating test suites. In the context of GUIs, a test case is a sequence of events to be executed which may detect faults in the application. However, a test case may be infeasible if one or more of the events in the event sequence are disabled or made inaccessible by a previously executed event (e.g., a button may be disabled until another GUI widget enables it). These infeasible test cases terminate prematurely and waste resources, so software testers would like to modify the test suite execution to run only feasible test cases. Current techniques focus on repairing the test cases to make them feasible, but this relies on executing all test cases, attempting to repair the test cases, and then repeating this process until a stopping condition has been met. We propose avoiding infeasible test cases altogether by predicting which test cases are infeasible using two supervised machine learning methods: support vector machines (SVMs) and grammar induction. We experiment with three feature extraction techniques and demonstrate the success of the machine learning algorithms for classifying infeasible GUI test cases in several subject applications. We further demonstrate a level of robustness in the algorithms when training and classifying test cases of different lengths.**

***Keywords*-GUI testing; software testing; event based testing; support vector machines; grammar induction; machine learning**

I. Introduction

As graphical user interfaces (GUIs) become nearly ubiquitous, researchers have proposed several methods for testing GUIs. One such method is a directed-graph based model [1] which supports automatic test case generation. The event-flow model captures all possible event sequences in the GUI; however, some event sequences may be prohibited by state-based constraints [2], and thus the event sequence which comprises a test case may be *infeasible*, meaning that the event sequence contains at least one event which 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.

Previous work by Huang et al. [2] used a genetic algorithm to repair these infeasible test cases. The work in the present paper takes an alternate approach and uses 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 paper we present two methods for predicting test case infeasibility: support vector machines (SVMs) [3] and induced grammars [4]. 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 [5] 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 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.

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 III-A we discuss three 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.

Our other approach is to induce grammars from the infeasible test cases and use them 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 [6], 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.)

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.

II. Background and Related Work

We present the following outline of background material on graph model-based GUI testing, and machine learning techniques and software testing.

A. 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 [7], but here we restrict our discussion to graphical models.

An *event-interaction graph* (EIG) is a model of a GUI which 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 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 walking the graph to produce sequences of events. However, the EIG generated for a GUI through GUI Ripping [1] may be only an approximation which 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 (i.e., there is at least one event which the model expects to be available but is actually unavailable/disabled at that moment during sequence execution). Memon offers more information about EIGs in [6].

Huang et al. [2] 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 which cause a test case to be infeasible may change as the software evolves, faults are fixed, and new faults are introduced.

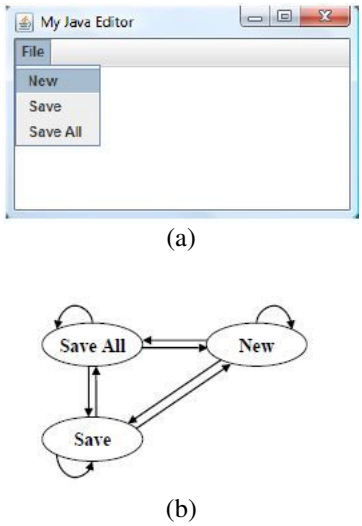

Figure 1. (a) A simple GUI. (b) The EIG for the GUI. Taken from Huang et al. [2].

B. Machine Learning and Software Testing

Many research endeavors, like the ones conducted by Hoefel and Elkan [8], have been performed using supervised machine learning algorithms. Several, like Chen et al [9], 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.

One common characteristic of many machine learning techniques is that some data pre-processing must be performed in order to provide the classifier appropriate features from which to learn. 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 [10], where pairwise sequence combination similarity and support vector machines are combined for homology detection.

Little work has been performed in the software testing field using machine learning [11]; especially for predicting the feasibility of test sequences. Briand et al. use decision trees 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 [12]. Briand [13] also outlines the use of machine learning for improved fault localization. Yilmaz et al. [14] use classification trees for testing system configurations. These models are built from a training set of system configurations with known failure/success outcomes.

Perhaps the most similar work is by Baskiotis et al. [11] who 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, we would like a general method to predict infeasible GUI test cases regardless of the underlying model that generated them.

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 [15], work on long sequences [16], learn from noisy data [17], learn a single grammar [18], use machine learning to refine other results [19], learn features from non-sequential data [4], or use algorithms that make assumptions incompatible with our data [20].

Grammar induction seems to be a promising approach for learning infeasible event sequences, although grammar induction is acknowledged to be a hard problem [5]. 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 $L(G)$.

C. 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.

D. Grammar Induction

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 [17], and induced Regular Grammars from XML documents to infer Document Type Definitions [18].

Another type grammar is the Context-Free Grammar (CFG), which is more expressive than Regular Grammars [21] and could conceivably capture a wide variety of GUI constraints. Some grammar induction techniques induce CFGs [22], 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 IV-A the constraints in our data can all be modeled using regular grammars which are a subset of CFGs [21].

III. Methods

A. 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 data modeling 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 three separate algorithms to generate three potential SVM input vectors which we call Basic, Pairwise, and Full Pairwise. We then compared the classification accuracy for each type of input vector, which we discuss in Section V-C.

Having this in mind, the generation of the Basic, Pairwise, and Full 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 sequence. 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
```

Basic, Pairwise, and Full Pairwise 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 IV-A). 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, or Full 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.

B. 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

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

Thus, we are essentially estimating the probability

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

```
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

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.

IV. Evaluation

Below we describe our test case data and evaluation methodology.

A. 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 [2].

The data is a set of test suites, where each test suite is a set of test cases. Each test case of length n is composed of a string of n 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 n). For example,

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

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 [23]. 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. [2] for a full description of the applications and the test case generation methodology.

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

B. 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.

We create five pairs of training data sets T_r and testing data sets T_t where for each pair $x\%$ of the test cases in the test suite T are in the training data and $(100-x)\%$ of the test cases are in the testing data (for $x \in \{10, 20, 30, 40, 50\}$). We generate T_r and T_t from test cases $TC \in T$ as follows:

1) Create the sets $I = \{tc \in T | tc \text{ is infeasible}\}$ and $F = \{tc \in T | tc \text{ is feasible}\}$.
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.

Other than the approach described above, no attempt was made 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.

C. Experimental Design

We evaluated our algorithms by addressing the following questions:

- **Question 1**: Determine the effect of training data set size on classification errors.
- **Question 2**: Determine whether a classifier trained on one test case length can 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.

D. 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% 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, and 5.

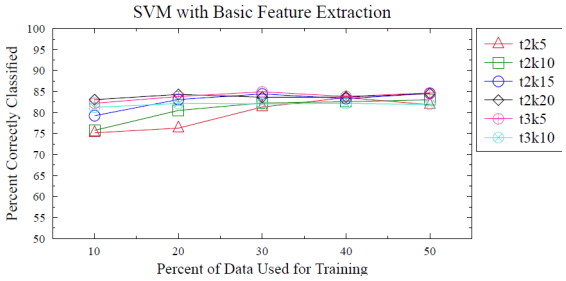

Figure 3. The plot shows the results for the average results of the Basic algorithm's performance.

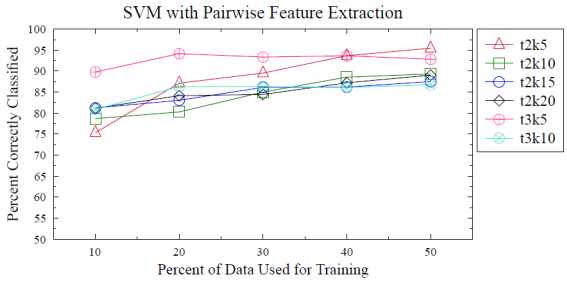

Figure 4. The plot shows the results for the average results of the Pairwise algorithm's performance.

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 6 shows the average results across all GUI applications for training and classifying t2k5 and t2k10.

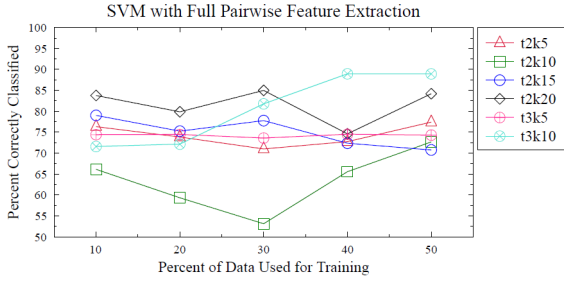

Figure 5. The plot shows the results for the average results of the Full Pairwise algorithm's performance.

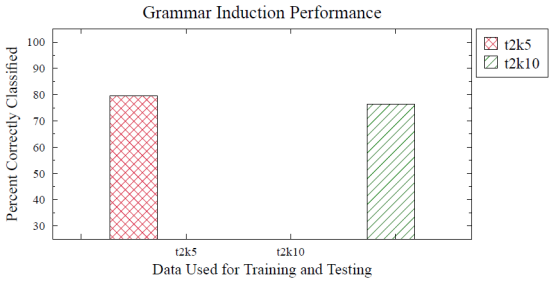

Figure 6. The average correct classification rate across all GUIs for induced grammars. The size of the training data is fixed at 10% of the data.

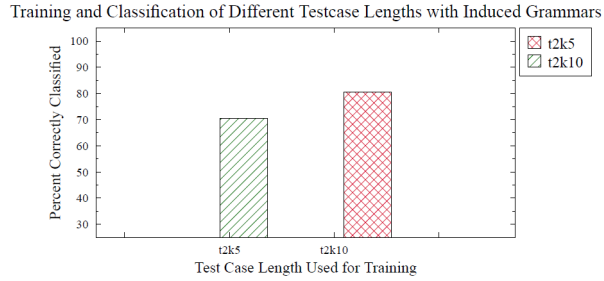

Figure 10. A graph of the average performance of the induced grammars for two covering arrays as training data vs. the remaining array as test data.

E. 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 7, 8, and 9. In all cases the SVMs used 30% of the data during the training phase and 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 10.

V. Results

A. 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 its performance and has no clear pattern as we change the size of the training data set. In all feature extraction techniques we mostly see correct classification rates at or above 75%.

The data collected for the induced grammars is limited, but from Figure 6 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.

B. 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. 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 10 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.

C. 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 paper 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.

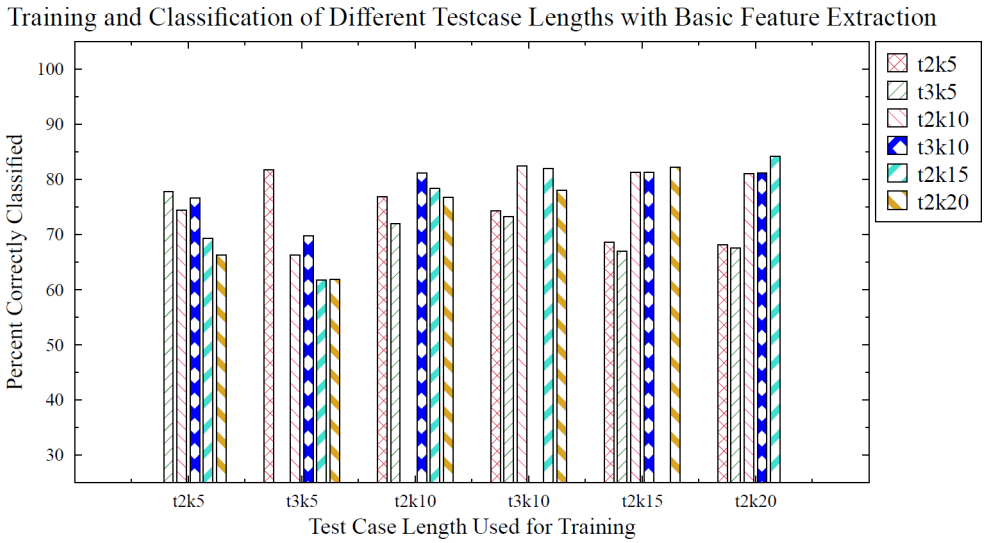

Figure 7. A graph of the average performance of the Basic algorithm for each covering array as training data vs. the remaining arrays as test data.

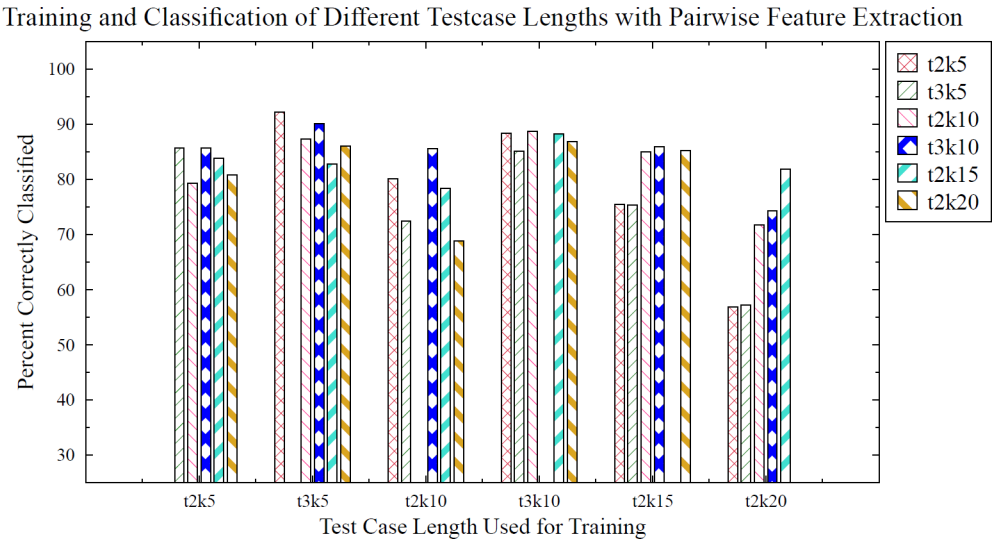

Figure 8. A graph of the average performance of the Pairwise algorithm for each covering array as training data vs. the remaining arrays as test data.

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 III-A, 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 $e^*(\# - e)^*$.

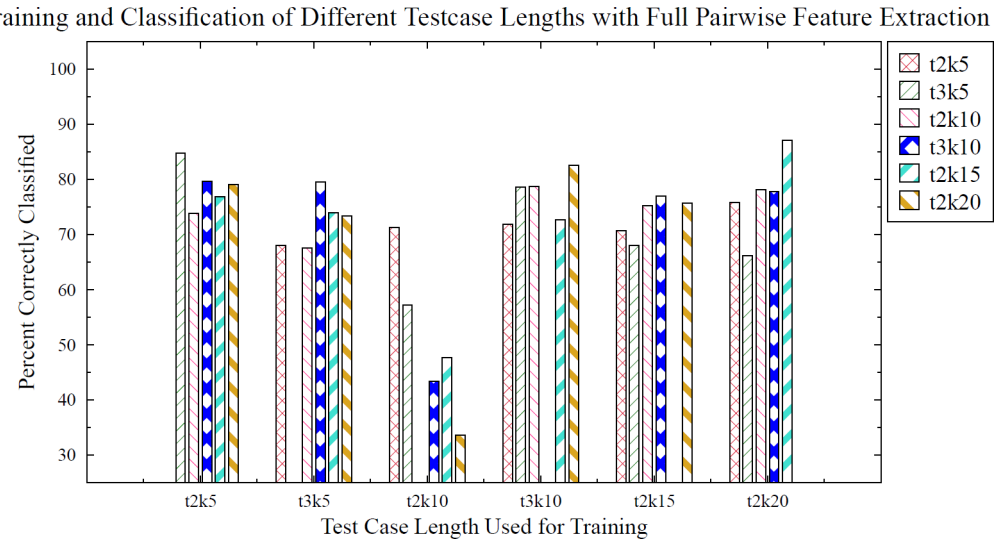

Figure 9. A graph of the average performance of the Full Pairwise algorithm for each covering array as training data vs. the remaining arrays as test data.

Fourth, 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.

VI. Conclusion

In this paper we have demonstrated the effectiveness of using two supervised learning algorithms 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] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 884–896, 2005.

[2] S. Huang, M. Cohen, and A. M. Memon, "Repairing GUI test suites using a genetic algorithm," in *Proc. 3rd IEEE Int. Conf. Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2010.

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

[4] J. Hipp, U. Güntzer, and G. Nakhaeizadeh, "Algorithms for association rule mining — a general survey and comparison," *SIGKDD Explor. Newsl.*, vol. 2, no. 1, pp. 58–64, 2000.

[5] S. Porat and J. A. Feldman, "Learning automata from ordered examples," *Machine Learning*, vol. 7, pp. 109–138, 1991, 10.1023/A:1022642911489. [Online]. Available: http://dx.doi.org/10.1023/A:1022642911489

[6] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[7] N. T. Ana Paiva, João C. P. Faria and R. F. A. M. Vidal, "A model-to-implementation mapping tool for automated model-based gui testing," in *Int. Conf. Formal Engineering Methods*, 2005, pp. 450–464.

[8] G. Hoefel and C. Elkan, "Learning a two-stage SVM/CRF sequence classifier," in *Proc. 17th ACM Conf. Information and Knowledge Management*. New York, NY, USA: ACM, 2008, pp. 271–278.

[9] H. Chen, H.-X. Zhou, X. Hu, and I. Yoo, "Classification comparison of prediction of solvent accessibility from protein sequences," in *Proc. 2nd Conf. Asia-Pacific Bioinformatics*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, pp. 333–338.

[10] L. Liao and W. S. Noble, "Combining pairwise sequence similarity and support vector machines for remote protein homology detection," in *Proc. 6th Annu. Int. Conf. Computational Biology*. New York, NY, USA: ACM, 2002, pp. 225–232.

[11] N. Baskiotis, M. Sebag, M.-C. Gaudel, and S. Gouraud, "A machine learning approach for statistical software testing," in *Proc. 20th Int. Joint Conf. Artifical Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 2274–2279.

[12] L. C. Briand, Y. Labiche, Z. Bawar, and N. T. Spido, "Using machine learning to refine category-partition test specifications and test suites," *Information and Software Technology*, vol. 51, no. 11, pp. 1551 – 1564, 2009, third IEEE International Workshop on Automation of Software Test (AST 2008); Eighth International Conference on Quality Software (QSIC 2008). [Online]. Available: http://www.sciencedirect.com/science/article/B6V0B-4WP47MR-1/2/e46ad361823030f7cb6d8407ab60101a

[13] I. Gondra, "Applying machine learning to software fault-proneness prediction," *J. Syst. Softw.*, vol. 81, no. 2, pp. 186–195, 2008.

[14] C. Yilmaz, M. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20 – 34, 2006.

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

[16] A. Giordana, P. Terenziani, and M. Botta, "Recognizing and discovering complex events in sequences," in *Proc. 13th Int. Symp. Foundations of Intelligent Systems*. London, UK: Springer-Verlag, 2002, pp. 374–382.

[17] U. Galassi and A. Giordana, "Learning regular expressions from noisy sequences," in *Proc. 6th Int. Symp. Abstraction, Reformulation and Approximation*, 2005, pp. 92–106.

[18] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, "Inference of concise regular expressions and DTDs," *ACM Trans. Database Syst.*, vol. 35, no. 2, pp. 1–47, 2010.

[19] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," in *Proc. 21st IEEE Int. Conf. Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 295–304.

[20] T. G. Dietterich, "Machine learning for sequential data: A review," in *Proc. Joint IAPR Int. Workshop Structural, Syntactic, and Statistical Pattern Recognition*. London, UK: Springer-Verlag, 2002, pp. 15–30.

[21] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusetts: Addison-Wesley, 1979.

[22] O. Unold, "Grammar-based classifier system: a universal tool for grammatical inference," *W. Trans. on Comp.*, vol. 7, pp. 1584–1593, October 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1486693.1486700

[23] X. Yuan, M. Cohen, and A. 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: IEEE Computer Society, 2007.