Learning Effective Oracle Comparator Combinations for Web Applications

Sara Sprenkle
Department of Computer Science
Washington & Lee University
Lexington, VA 24450
sprenkles@wlu.edu

Emily Hill and Lori Pollock
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
{hill, pollock}@cis.udel.edu

Abstract

Web application testers need automated, effective approaches to validate the test results of complex, evolving web applications. In previous work, we developed a suite of automated oracle comparators that focus on specific characteristics of a web application's HTML response. We found that oracle comparators' effectiveness depends on the application's behavior. We also found that by combining the results of two oracle comparators, we could achieve better effectiveness than using a single oracle comparator alone. However, selecting the most effective oracle combination from the large suite of comparators is difficult. In this paper, we propose applying decision tree learning to identify the best combination of oracle comparators, based on the tester's effectiveness goals. Using decision tree learning, we train separately on four web applications and identify the most effective oracle comparator for each application. We evaluate the learned comparators' effectiveness in a case study and propose a process for testers to apply our learning approach in practice.

1 Introduction

The critical need for reliable web applications, coupled with their frequent code modification [13], motivates developing automated, accurate validation approaches, specialized for web applications. In previous work, we developed a suite of general, automated oracle comparators that each focus on specific characteristics of a web application's HTML response [19]. We found that the oracles' effectiveness in terms of correctly identifying failures and in minimizing the incorrect identification of correct test results as failures depends on the application's behavior. We also found that by unioning the results of carefully selected individual oracle comparators, we sometimes achieved better effectiveness in identifying failures than using an individual comparator alone. While the simple union combination was promising, a tester typically does not have the time and resources to evaluate all the comparators, select which of the individual oracle comparators to combine (22 comparators in our case), and determine how to combine the oracle results (e.g., union or some other operation) to create the most effective oracle comparator for the application under test.

In this paper, we propose applying decision tree learning to systematically select the best oracle combination. We applied decision tree learning because, unlike other machine learning approaches such as neural networks, decision tree learning outputs an intuitive, generalized model of a training data set that can be easily interpreted by the tester. Furthermore, decision tree learning is tunable to the tester's effectiveness goals, and it executes quickly—less than a minute to train on 1.4 million responses.

We applied decision tree learning to determine the best oracle comparators for four subject applications. While some of the learned oracles matched our intuitions, others were surprising. In some cases, an individual oracle comparator had better effectiveness than any oracle combination. Our results do not indicate one oracle that fits every application. Thus, we suggest that, if an application has similar behavior to one of our subject applications, the tester can select the appropriate, learned oracle. However, for the best effectiveness, a tester should follow our proposed process for learning an oracle by training on the application.

After Section 2 summarizes the oracle comparators and results from our previous work, Section 3 presents our approach to learning the best oracle combination. In Section 4, we describe our case study to evaluate the oracle combinations and present and analyze our results. We present related work in Section 5 and conclude with our recommended process for testers to apply our approach and future work in Section 6.

2 HTML-based Oracle Comparators

In previous work, we implemented a suite of 22 HTML-based oracle comparators that focused on various charac-

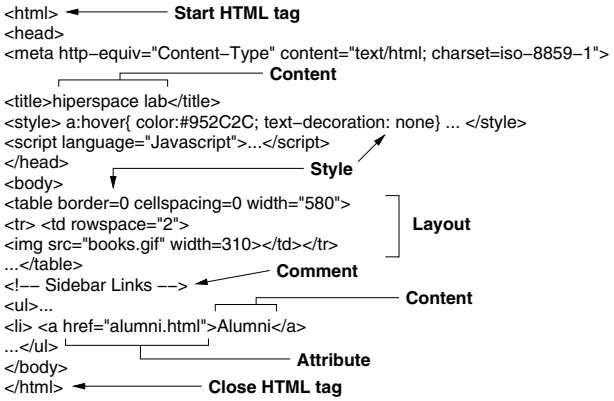

Figure 1. Simplified Excerpt of HTML File

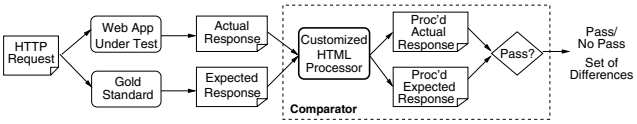

Figure 2. HTML-based Oracle Comparator Validation Process

teristics of a web application's HTML response [19]. We summarize the contributions of that work in this section.

HTML (Hypertext Markup Language) [10] is the standard language for publishing web pages, regardless of the programming language(s) used to implement the web application. As shown in Figure 1, authors include content as well as HTML elements to structure the document and provide presentation directives (e.g., headings, hyperlinks, style, and layout) in an HTML document. HTML elements usually have a start tag with zero or more attributes (name/value pairs) and may have an end tag and content enclosed between tags.

Any part of an HTML document may contain failures. An oracle comparator that naively detects every difference between the actual and expected HTML output could mistakenly report a failure when the difference lies in real-time, dynamic information. Reporting such *false positives* can overwhelm a tester if there are many false positives and can waste developer effort tracking down nonexistent bugs. In contrast, an oracle comparator that focuses on specific internal details of behavior may miss reporting faults (*false negatives*), resulting in heavy penalties such as loss of consumer confidence and business revenues.

Figure 2 illustrates the implemented comparators' validation process. Each oracle comparator performs customized processing on the expected and actual HTML response from an HTTP request and outputs if the request *passed* and, if the request did not pass, the set of differences between the responses to help identify the fault. Figure 3 illustrates the partial ordering of the 22 implemented oracle comparators based on the information from the HTML document that each oracle comparator uses; we use the abbreviations in bold when reporting results in this paper. The benefits and limitations of the oracle comparator classes (**Document**, **Content**, **Tags**, **Tags+ImptAttrs**, **TagNames**, **Forms**, and **UnorderedLinks**), italicized in Figure 3, are summarized in Table 1. Details about the oracle comparators and justifications for our design decisions are in [19].

The results of our empirical evaluation of the comparators' effectiveness indicated that the application's behavior (whether the application has nondeterministic or real-time behavior) affects the effectiveness of the oracle comparators. We also found that, in general, the effectiveness trends followed the partial ordering: comparators at the top of the partial ordering had the fewest false negatives and the most false positives, while oracles towards the bottom had more false negatives and fewer false positives. Since the **Document** oracle is the root of the partial ordering, if **Document** reports "pass", all the child oracles will also report pass; conversely, **Document** will report a failure if any other comparator reports a failure. Because of this tradeoff between false positives and false negatives, we investigated if unioning or intersecting oracle comparators (such as one that reports few false positives with one that reports low false negatives) created a more effective oracle. We found that unioning carefully selected comparators had better effectiveness than either individual comparator and that, across all applications, N+I ∪ C-WD had the best overall effectiveness.

While N+I ∪ C-WD was effective, we did not exhaustively combine multiple oracles or use alternative operations to combine the comparators to identify the optimal oracle combination. In this paper, we explore a systematic approach that quickly learns the most effective oracle combination.

3 Learning Effective Oracle Combinations

We applied *decision tree learning* to identify the most effective combinations of oracle comparators. Decision tree learning is an inductive learning method that constructs a classifier, or decision tree, for a given training data set [15]. Our classification problem is to correctly classify an HTTP request as *pass* or *no pass* (whether the resulting response exhibits a failure or not), given a *feature set* containing the results of several oracle comparators and a quantification of application behavior. A decision tree is trained on this set of features—the actual *pass* or *no pass* output of each oracle comparator and a quantification of application behavior—and an expected *pass* or *no pass* classification for each request/response pair in the training set. The learned decision tree is our effective oracle combination.

The remainder of this section describes our training data

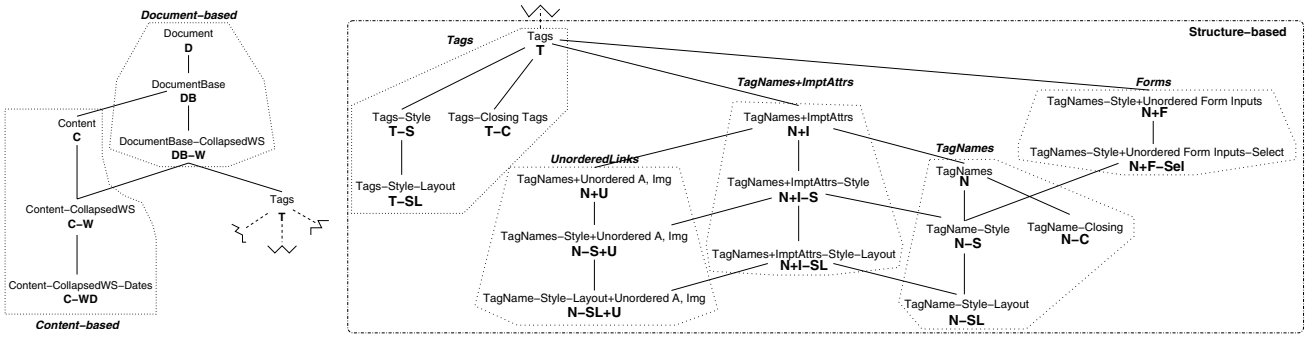

Figure 3. Partial Ordering of Implemented Oracle Comparators

Comparator Class	Processes	Benefits	Drawbacks
Document	HTML document	Identifies all failures in HTML document	Reports false positives for nondeterminism, real-time behavior in content and tags
Content	Text enclosed between tags	Identifies all failures in content	Misses structure-only failures; reports false positives for real-time behavior in content
Tags	HTML tags	Identifies all failures in tags	Misses failures in content; reports false positives for real-time behavior in tags
Tags+ImptAttrs	HTML tags and "important" attributes	Identifies failures that occur in the important attributes	Misses failures in content and in "unimportant" attributes
TagNames	Names of the tags only	Identifies failures that occur in the tags	Misses failures in content and the tags' attributes
Forms	Tag names, specialized form handling	Identifies failures in forms accurately—not because order of form inputs changed	Misses failures in content and when form input order matters
UnorderedLinks	Tag names, specialized link and image handling	Identifies failures in tags accurately—not because link or image order changed	Misses failures in content and when link or image order matters

Table 1. Qualitative Comparison of Oracle Comparator Classes

in more detail, our methodology for learning the oracle comparator combinations, and the learned combinations.

3.1 Training Data

Our training data consists of the actual *pass* or *no pass* output from each oracle comparator, a quantification of application behavior, and the human-generated expected oracle results. To generate both *pass* and *no pass* expected oracle results for training, we executed test suites on fault-seeded versions of four subject applications.

3.1.1 Subject Applications and Test Suites

The four subject applications have varying sizes (1K-50K non-commented lines of code), technologies, and representative web application activities: a conference website (Masplas); an e-commerce bookstore (Book) [8]; a course project manager (CPM); and a digital library (DSpace) [6]. Table 2 summarizes the applications' code characteristics.

Application	Classes	Methods	Statements	NCLOC
Masplas	9	42	441	999
Book	11	385	5250	7791
CPM	75	172	6966	8947
DSpace	274	1453	27136	49513

Table 2. Subject Application Characteristics

We generated test suites by deploying each subject application, collecting user requests to each application, and then converting the requests into a test suite of user sessions [17].

Table 3 shows characteristics of each application's test suites. The number of requests is the total number of HTTP requests. We show both the number of requests and the number of HTML responses because some DSpace responses are the publication itself in various formats, and therefore a simple `diff` of the publications—rather than an HTML comparator—can validate the response. The test suites do not completely cover the code because users do not access some code, such as error and administrative code and

Application	Test Input # Test Cases	# Requests	% Stmt Coverage	Test Output # Responses	Seeded Faults	Exposed Faults
Masplas	169	1103	90.5%	1103	28	22
Book	125	3564	56.8%	3564	39	36
CPM	890	12352	78.4%	12352	135	96
DSpace	75	3183	51.6%	3023	50	20

Table 3. Test Suites, Responses, and Faults

code for alternative configurations. In previous work [19], we found that our test suites exercised only deterministic behavior for Masplas and Book and exercised some real-time and nondeterministic behavior in CPM and DSpace.

3.1.2 Creating Faulty Application Versions

To create faulty versions of each subject application, students familiar with web applications manually seeded faults into each application. We also seeded naturally occurring faults that were discovered by users during application deployment into Masplas, CPM, and DSpace. In general, five types of faults were seeded—data store (faults that exercise application code interacting with the data store), logic (application code logic errors in the data and control flow), form (e.g., modifications to name-value pairs and form actions), appearance (faults that change the way the page appears), and link (changes to hyperlinks). A seeded fault can be in multiple categories. The rightmost column in Table 3 reports the number of seeded and exposed faults. Since the test suites did not exercise some seeded faults and some faults did not manifest in the HTML responses, we report the number of seeded and exposed faults separately and only analyze the comparator results for exposed faults.

3.1.3 Expected and Actual Oracle Results

For our training data, we need the expected and actual oracle comparator results for a set of responses. To generate these results, we replayed the test suite *with_state* [18], where application state is restored before replaying every test case in the suite on the clean and fault-seeded versions of each subject application, within our experimental framework described in previous work [18]. By replaying *with_state*, test suite execution on the faulty versions closely matches the clean execution, but faults that manifest themselves in the application's state will not propagate in the state for subsequent test cases to expose. If we allowed faulty state to propagate, the responses would diverge from the clean responses more, and failures would become trivial to detect. Replaying *with_state* reduces the number of responses that may display a failure, making it more difficult for comparators to detect failures and easier to differentiate between the comparators. We then executed each oracle comparator, comparing the test suite's responses from the clean version of the application (the expected test results) with the responses from the fault-seeded versions (actual results). We manually determined the *expected* pass/no pass results for the ideal oracle by checking if each response exhibited a failure.

Application	Pass Trivial	Non-Trivial	No Pass	Total
Masplas	22924	1208	134	24266
Book	107990	18764	1550	128304
CPM	1164661	19449	1682	1185792
DSpace	53356	5651	4653	63660
Total	1348931	45072	8019	1402022

Table 4. Expected Oracle Results for Training Data

Table 4 summarizes the expected oracle results for our training data. The majority of the training data were passes; furthermore, most of the passes are *trivial*, where the responses from the clean and fault-seeded versions have no differences (i.e., **Document** outputs pass) because faults do not manifest themselves in the majority of responses. For trivial passes, any HTML comparator will correctly report pass. We report trivial and non-trivial passes separately because differentiating between comparators when including trivial pass results becomes more difficult.

3.1.4 The Role of Application Behavior

As reported in previous work [19], application behavior such as nondeterminism and real-time behavior plays a role in the effectiveness of the oracle comparators, and therefore we include it as a feature of our training data. For example, **Document** effectively identifies failures in deterministic applications but reports too many false positives for innocuous differences from nondeterministic applications. Thus, in addition to the output of each oracle comparator and the human-annotated expected results, our training data set includes two binary features that quantify application behavior: application nondeterminism and response nondeterminism. Because a tester can automatically discover which application responses are nondeterministic, we believe that including these features does not put an undo burden on the tester.

We automatically determined which application re-

Application	Oracle	False -	False +	Total
Masplas	DB-W	0	0	0
Book	T ∪ C	4.25	0	4.25
Book	D	0	7.63	7.63
CPM	dD ∪ N+F-Sel	0.43	0	0.43
CPM	D	0	7.96	7.96
DSpace	N+U	0.45	0.04	0.49
DSpace	N-S+U	0.82	0	0.82
DSpace	C-WD ∪ N+F	0	5.07	5.07

Table 5. Percent Error for Oracle Comparators Trained on Each Application Separately

Application	Oracle	False -	False +	Total
Masplas	DB-W	1.73	12.24	13.97
Book	T ∪ C	0	19.32	19.32
Book	D	0	19.73	19.73
CPM	dD ∪ N+F-Sel	5.70	5.38	11.08
CPM	D	0	19.83	19.83
DSpace	N+U	25.56	0	25.56
DSpace	N-S+U	25.56	0	25.56
DSpace	C-WD ∪ N+F	17.86	3.75	21.61

Table 6. Percent Error for Oracle Comparators Cross-validated on Other Three Applications

sponses are nondeterministic by executing the test suites on the clean version of the application (no seeded faults) nine times over several months. Responses that are the same for each execution are considered deterministic, whereas responses that change at least once are considered nondeterministic. Masplas and Book are entirely deterministic applications, whereas CPM and DSpace have both deterministic and nondeterministic and real-time responses.

3.2 Learning Oracle Combinations

This section describes our methodology for learning the most effective oracle combinations. We used the C5.0 decision tree learner (`http://www.rulequest.com/see5-info.html`) to construct a decision tree, learned from our training data. To allow the tester to apply the learned oracle combinations easily in practice, we interpret the decision tree that was output by C5.0 as a propositional logic formula. We converted a decision tree into a logic formula by rewriting the tree path to a no-pass classification as a disjunction of conjunctions of the constraints [12]. We then applied basic logic operations to simplify the resulting oracle combinations.

C5.0 supports variable misclassification costs, i.e., we can tune C5.0 to generate oracle combinations with fewer false positives, false negatives, or total error. Variable misclassification costs are useful because, in practice, a tester might want to use an oracle combination with no false positives to quickly isolate severe faults. Once these easily detected faults have been fixed, a tester could apply an oracle with few false negatives for more comprehensive testing. Therefore, we execute C5.0 several times, specifying various misclassification costs, to create oracle combinations with different error rates.

Interestingly, decision trees trained on all four applications created overly complex oracle combinations that did not agree with our intuition of appropriate combinations of oracle comparators. Because such complicated trees may not accurately represent a generalized model of the training data, we only report here the oracle combinations derived specifically from each application's training data. Table 5 shows the percent of false positives and false negatives for each learned oracle on its training data. Note, we report the false positive rate as

$$\frac{\#\ reported\ failures}{\#\ failures\ Document\ reports}$$

so that we do not include trivial passes—cases where the expected and actual responses have no differences—to better distinguish between comparators.

Masplas. Because Masplas does not exhibit any real-time or nondeterministic behavior, the **Document**-based oracle DB-W is sufficient to achieve zero false positives and zero false negatives, as shown in Table 5.

Book. Although Book is deterministic, unlike Masplas, no combination of oracles yields zero false positives and false negatives. The best overall combination for Book is T ∪ C. T ∪ C is very similar to DB-W (the same oracle learned for Masplas), but DB-W increases the error rate from 4.25 percent to 4.41 percent. To eliminate false negatives at the cost of false positives, D is the best oracle for Book.

CPM. The best oracle combination for CPM takes advantage of the nondeterministic response variable and applies the D oracle only to the deterministic responses. We call this new oracle combination dD for a deterministic document oracle. Because many of CPM's responses contain forms, the N+F-Sel is also applied in CPM's best oracle: dD ∪ N+F-Sel. As shown in Table 5, CPM's best oracle combination has no false positives. Like Book, the D oracle is sufficient to eliminate false negatives for CPM.

DSpace. The best overall oracle for DSpace is N+U, which has very low false positives and false negatives. N+U performs best for DSpace because there is nondeterminism in the order of links in a response. To eliminate false positives at the expense of almost doubling the number of false negatives, the N-S+U oracle can be used instead. As shown in Table 5, eliminating the false negatives by applying the C-WD ∪ N+F oracle combination causes the number of false positives to increase.

Cross-validation. Table 6 shows the percent of false posi-

tive and false negative results for each oracle cross-validated on the data for the remaining three applications. Low total error, such as the error for dD ∪ N+F-Sel and DB-W, implies the oracle combination is successful on its training data as well as on new data. High total error implies that the oracle combination fits the training data well but does not generalize to other applications.

4 Evaluating Learned Oracle Combinations

This case study investigates the research question: *How many failures does a learned comparator miss (false negatives) and how many responses are incorrectly identified as failures (false positives) when used on data beyond the training set?*[1] To answer our research question, we evaluated the learned oracle comparators in two experiments. In the first experiment, we perform a comprehensive analysis of the effect of real-time and nondeterministic application behavior on the effectiveness of the oracle comparators. In the second experiment, we evaluate the oracle comparators in detecting failures caused by seeded faults.

Independent and Dependent Variables. The *independent variable* is the learned oracle comparator. Specifically, we evaluate the C-WD ∪ N+I, the oracle combination that had the best mean effectiveness in previous work [19], and the oracle combinations developed from our analysis in Section 3: DB-W, T, D, dD ∪ N+F-Sel, N+U, N+U-S, and C-WD ∪ F. The *dependent variable* is the failure detection error rate in terms of both false positives and false negatives.

Subject Applications and Test Suites. We used the subject applications and test suites described in Section 3.1.1. In the first experiment, we executed a larger DSpace test suite, containing 1800 test cases, 22129 requests, and 17892 responses with 65.6% statement coverage.

4.1 Methodology

Effect of Nondeterminism/Real-Time Behavior. For CPM and DSpace (our applications that exhibit nondeterministic and real-time behavior), we replayed the same test suite nine times on the clean version of the application over several months. We then executed each individual and combination oracle comparators pairwise on the responses from the nine executions of each test suite, totaling 36 outputs for each comparator per application[2]. Because the applications do not contain faults during this experiment, any failures that an oracle comparator reports are false positives. For CPM, 2964 of the 444,672 paired responses (less than 1%) were different, as determined by the **Document** comparator. Of the 1,240,344 pairs of DSpace responses, 48,458 (3.9%) were different.

Failure Detection Effectiveness. We analyzed each learned oracle comparator's effectiveness at identifying failures from seeded faults in all our subject applications. The methodology for seeding faults was presented in Section 3.1. Since we used considerably more results for CPM than the other applications, which may bias the results towards the comparators learned from CPM, we also evaluated the comparators on normalized data, where we duplicated Masplas's, Book's and DSpace's results 49, 10, and 19 times, respectively, so that the number of results were each approximately equal to CPM's and then combined them with CPM's results.

4.2 Threats to Validity

Since we performed our study with four applications, a study with additional applications may be necessary to generalize the results; however, we chose subjects with different technologies, implementers, and application and usage characteristics to help reduce the threat to generalizing our results. We believe that our oracle comparators are general and will handle most web applications and their HTML output; however, we designed our oracle comparators with knowledge of how our subject applications behave.

Manually seeded faults may be more difficult to expose than naturally occurring faults [2]. Although we tried to model the seeded faults as closely as possible to naturally occurring faults—even including naturally occurring faults from previous deployments, some of the seeded faults may not represent natural faults. Because the faults may not be realistic, we may bias the results to some comparator because it does or does not detect those types of faults. Since we only consider the number of HTML responses containing manifestations of the fault detected/missed by the oracles and not the severity or detectability of the faults detected/missed, the conclusions of our experiment could be different if the results were weighted by fault severity or detectability. We also do not analyze sources of false positives beyond nondeterminism or real-time behavior.

4.3 Results and Analysis

4.3.1 Effect of Nondeterminism/Real-time Behavior

Table 7 presents the false positive results for the first experiment to partially answer our research question. Recall that any failure (difference in expected and actual processed responses) reported by an oracle is a false positive because the application is the clean version. We report the false positive rate as defined in Section 3.2.

[1] Our evaluation does not include detecting unmet time/space-constraint failures.

[2] We chose nine executions and 36 suite-level outputs by the comparator to obtain a good sample size for indicating nondeterminism.

Oracle Comparator	False Positive Error	
	CPM	DSpace
N+U	0	20.44
N-S+U	0	9.36
dD ∪ N+F-Sel	38.7	10.9
C-WD ∪ N+F	98.8	37.6
N+I ∪ C-WD	100	51.8
T ∪ C	100	100
DB-W	100	100
D	100	100

Table 7. Effect of Nondeterminism/Real-time Behavior on Comparators

Since the test suites were replayed over several months, the real-time behavior has more of an effect on the responses than in the training data used to develop the oracles. Thus, even the oracle combinations developed to minimize false positives for a specific application, while still maintaining low false negatives, report some false positives: CPM's dD ∪ N+F-Sel reported 38.7% of responses with differences as failures and DSpace's N-S+U reported 9.35% of the differences as failures. The oracle combinations developed for a deterministic application (T ∪ C) and across all applications (N+I ∪ C-WD) have much higher false positive errors than the oracle comparators designed specifically for the applications with nondeterministic and real-time behavior.

4.3.2 Failure Detection

Figure 4 reports the oracle comparators' error rate in detecting failures across all applications, sorted by total error. For the oracle comparator labels, we used "*" instead of ∪. The false positive rate (in light gray) is stacked on top of the false negative rate (in dark gray) to show the total error. Again, the error rates do not include the responses that trivially pass (described in Section 3.1.3); to calculate the error rates, we divide the comparator's number of reported false positives or false negatives by the number of failures reported by the **Document** comparator. Excluding the responses that trivially pass does not change the relative performance of the comparators but helps to distinguish the comparators. When we include the responses that trivially pass, all the reported oracles have error rates less than 1.5%.

The best oracle comparator combinations in terms of least total error are dD ∪ N+F-Sel (the oracle developed to minimize error for CPM) and N+I ∪ C-WD, the oracle combination identified as the best in previous work [19]. As expected from the cross-validation results in Table 6, dD ∪ N+F-Sel performed well when applied to all the applications. Oracles with high cross-validation error rates also had more error as shown in Figure 4. The **Document**-based

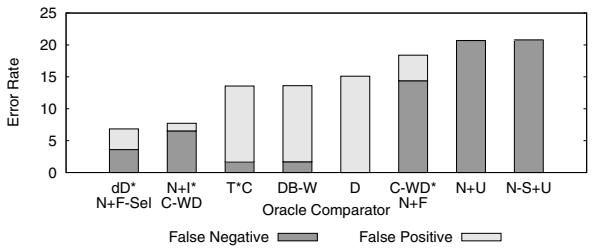

Figure 4. Error Rates Across All Applications

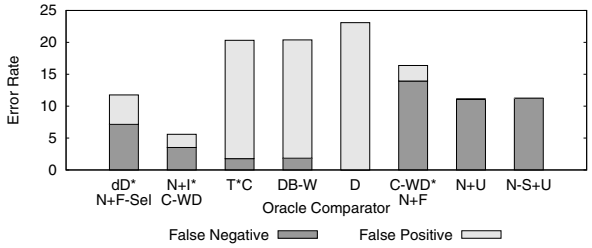

Figure 5. Error Rates Normalized Across Applications

oracles have the least false negatives at the cost of more false positives.

We also evaluated the comparators on normalized data, as described in Section 4.1. Figure 5 presents the error rates on the normalized data for the comparators, sorted in the same order as Figure 4. N+I ∪ C-WD has the best overall effectiveness on the normalized data.

Not surprisingly, Figures 4 and 5's results indicate the most effective oracles in general examine both an HTML response's content and structure because failures can manifest in any part of a response.

5 Related Work

Researchers have developed various solutions to the challenges of oracle comparators, including pseudo oracles [20], automating oracles [11, 16], and creating specialized oracles for specific domains using model-based techniques, e.g., [1, 4, 5, 11]. Our comparators are automated, consistent pseudo oracles (focused on a subset of the web applications' output—HTML responses) that do not require a specialized model. Other groups have mentioned using HTML-based oracles [5, 7] but provided few details on the comparator or experimental investigation into the accuracy of their oracles in the context of nondeterministic behavior.

Other researchers have applied machine-learning techniques to software engineering and, specifically, classifying program executions as pass or no pass [3, 9, 14]. However,

rather than learning from combinations of oracles, these approaches use execution profile information for features.

This paper differs from our previous work [19] by providing a systematic, learning-based approach to selecting an oracle comparator for web applications.

6 Conclusions and Future Work

In this paper, we proposed a general decision tree learning approach to identify the most effective oracle combinations for web application testing. We applied this approach to four applications and evaluated the learned oracle combinations in two experiments. Our results indicate that the most effective oracle for a given application is learned by training on its own expected oracle results. Across all applications, the best oracle comparator is N+I ∪ C-WD. The most effective learned oracle comparator is dD ∪ N+F-Sel, which has fewer false positives for nondeterministic applications than N+I ∪ C-WD.

More importantly, to learn an oracle specialized to an application, we recommend the following process to testers:

1. Create oracle training data. Since our results do not support creating an oracle combination based on training data derived from another application, a tester should execute a test suite on the clean version of her application as well as on faulty versions, using faults from bug reports. She should then execute the suite of individual oracle comparators on the clean and faulty responses and manually determine if each response passes (i.e., create the expected results). The tester should also identify the responses that have nondeterministic and real-time behavior by executing the test suite on the clean version multiple times and comparing the results. The tester combines the information about application behavior, actual comparator results, and expected comparator results to create the training data. In this paper, we used at least 20,000 responses to learn the oracle combination; however, we believe that the decision trees can learn effective oracles from fewer observations.

2. Use decision trees to learn the best oracle combination. The tester can tailor learning to meet her goals; for example, the tester may want the learned combination to reduce overall error, false positives, or false negatives.

3. Evolve oracle combination as application evolves. As the application evolves, the application's behavior and what constitutes a failure may change. The tester should repeat steps 1 and 2 with more recent bug reports to create a more accurate oracle.

In the future, we would like to further evaluate our decision tree learning approach by executing the learned oracle combinations on additional faults for the applications they were trained on, as well as on additional applications with different application behaviors.

References

[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software Systems and Modeling*, 4(2):326–345, April 2005.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Int'l Conf on Software Engineering*, 2005.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Int'l Symp on Software Testing and Analysis*, 2004.

[4] D. Chays, Y. Deng, P. Frankl, S. Dan, F. Vokolos, and E. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 14:17–44, Mar. 2004.

[5] G. DiLucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *Int'l Conf on Software Maintenance*, Oct. 2002.

[6] DSpace Federation. `http://www.dspace.org/`, 2007.

[7] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user session data to support web application testing. *Trans on Software Engineering*, 31(3):187–202, May 2005.

[8] Open source web applications with source code. `http://www.gotocode.com`, 2003.

[9] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouché. Techniques for classifying executions of deployed software to support software engineering tasks. *Trans on Software Engineering*, 33(5):287–304, May 2007.

[10] HTML 4.01 Specification. `http://www.w3.org/TR/html4/`, Dec. 1999.

[11] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Int'l Conf on Automated Software Engineering*, Oct. 2003.

[12] T. M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[13] J. Offutt. Quality attributes of web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, March/April 2002.

[14] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Int'l Conf on Software Engineering*, 2003.

[15] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[16] D. Richardson. TAOS: testing with analysis and oracle support. In *Int'l Symp on Software Testing and Analysis*, 1994.

[17] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Automated Software Engineering Conf*, Sept. 2004.

[18] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and fault detection for web applications. In *Int'l Conf on Automated Software Engineering*, Nov. 2005.

[19] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *Int'l Symp on Software Reliability Engineering*, Nov. 2007.

[20] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–70, 1982.