Syntactic Regression Testing for Tree-Structured Output

Elizabeth Soechting
University of Virginia
eas2h@virginia.edu

Kinga Dobolyi
University of Virginia
dobolyi@virginia.edu

Westley Weimer*
University of Virginia
weimer@virginia.edu

Abstract

Regression testing is used by software developers to ensure that program modifications have not negatively impacted the correctness of code. While regression testing has been successfully applied in many domains, programs such as web applications, XML processors, and compilers remain expensive to test because harmless program evolutions make the tests appear to fail: in our experiments 82% of test case output differences are false positives.

We present an automated tool that measures syntactic differences in the tree-structured output of such programs to reduce the number of false positives in, and thus the cost of, regression testing. We model test case outputs that merit human inspection through a set of structural and domain-specific features. We evaluate the performance of our technique on over 20,000 test case output comparisons, and find that we are three times as accurate as a naive comparator.

1. Introduction

In software engineering, regression testing is a way of life: after a program is changed to introduce features or remove defects, regression testing provides assurance that those modifications have not negatively impacted correctness. Regression testing is a major part of software maintenance [21]. Maintenance activities consume 70% [4] to 90% [26] of the total lifecycle cost of software, summing to over $70 billion per year in the United States [23, 32], with regression testing accounting for as much as half of this cost [12, 24]. Nevertheless, a lack of resources often restricts developers to utilize only a fraction of the available regression tests [8, 16, 35]. Consequently, even mature software projects are constrained to ship with known bugs [14].

*This research was supported in part by National Science Foundation Grant CNS 0716478, Air Force Office of Scientific Research grant FA9550-07-1-0532, and NASA grant NAS1-02117, as well as gifts from Microsoft Research. The information presented here does not necessarily reflect their positions or policies.

Regression testing is frequently limited by the effort required to compare results between two versions of program output. Formally, it can be viewed in terms of an *oracle* mechanism that produces an expected result and a *comparator* that checks the actual result against the expected result [3]. In practice the oracle is commonly taken to be the output of a previous, trusted version of the code on the same input and the comparator is a simple `diff` of the two outputs. Any difference implies that the test should be inspected by developers; this commonly suggests an error in the new version but may also indicate a discrepancy in the oracle output (e.g., the correct output may legitimately change as the program gains new functionality). Unfortunately, traditional regression testing is particularly burdensome for programs with tree-structured output (e.g., [29]) because using `diff` as the comparator produces too many false positives [31].

We propose SMART, a tool for syntactic regression testing, to reduce the cost of regression testing for programs with tree-structured output by providing a precise comparator. Programs that produce tree-structured output, such as XML, HTML, or abstract syntax trees, require a comparator that captures richer information. Consider new versions of: (1) a web application that produces HTML with a different copyright notice; (2) a program business logic core that serializes user sessions to XML with a different attribute order; and (3) a compiler that also includes additional debugging information. Even if no new defects have been introduced, a direct `diff` of the output will always report a potential error in all three cases, and thus always demand developer effort to either fix the program or update the oracle answer. The more the program evolves, the greater this unnecessary burden of regression test inspection will become. Nevertheless, the move toward XML- and HTML-emitting web-based applications continues at a rapid rate, and evolving and testing them remains critically important [38]. For such web applications, even a partial loss of functionality can cost businesses millions of dollars per hour [19]. Instead of just using character-based comparison through `diff`, we suggest that such projects use test comparators that understand the syntax and semantics

of tree-structured output. We propose to combine insights from structural differencing algorithms (e.g., [1]) as well as semantic features (e.g., [29]) into a distance model for test case output. This distance metric then forms the heart of a precise comparator, where a regression test should be inspected if the new output's distance from the oracle output exceeds a certain cutoff. We automatically determine which differences between program outputs and oracles are worth inspecting, based on their structural and syntactic features. Our technique is applied by inspecting a small fraction of the regression testing output (as one normally does), and using that information to train a model based on our features; the model dictates which of the remaining test outputs should be inspected. Our comparator is successful if it reduces false positives (i.e., correctly tells developers not to inspect test cases that have small changes but do not indicate defects) while minimizing or avoiding false negatives (i.e., incorrectly tells developers not to inspect real defects).

The contributions of this paper are:

- A set of features, both structural and syntactic, that help to determine if tree-structured regression test output should be inspected by human developers
- An experimental evaluation of a model and distance metric using those features; SMART is more than three times as accurate as `diff` averaged over 20,000 test case output pairs
- A quantitative and qualitative comparison of the relative power of those features, with a discussion of the possible impact on regression test practices

2 Motivating Example

Web application development has certain features that make comprehensive testing both difficult and important. Web applications have become an integral part of the global economy, with Internet-based e-commerce projected to reach over one trillion dollars by 2010 [28]. The short time-to-market horizon for web applications argues that testing should be a high priority, but in practice testing is often considered to be high-cost and low-benefit [9]. Compounding the problem are the evolving nature of user needs, the pressure to change, the complexity of web applications [22], and high quality-of-service requirements [37].

Testing web applications is challenging because they are often subject to updates that may not change the appearance or functionality experienced by the end user. Consider the `diff` output from two TXT2HTML test case versions [33]:

```
1  < <P>The same table could be indented.
2  < <TABLE border="1">
3  ---
4  > <p>The same table could be indented.</p>
5  > <table border="1" summary="">
```

In addition to including a new **`summary`** attribute, the newer output also handles the paragraph tag **`<p>`** differently. A single paragraph tag **`<p>`** is equivalent to the open-close pair of paragraph tags **`<p></p>`** for most browsers, with the distinction being that future versions of HTML will not support unmatched tags. A direct comparison would yield a false alarm (i.e., incorrectly instruct the developer to inspect the output) for this test case, because both of these changes represent updates to the HTML rather than errors. In general, the structure of web output is generated by the application, but the content itself often comes from a user or a database beyond the program's control. Consider a news site, such as CNN or the BBC: the content text changes daily, but that does not indicate a bug in the underlying web application.

The above example demonstrates that using `diff` to compare outputs for regression testing is often not appropriate for XML or HTML applications because of the potential for frequent false positives. Alternatively, developers may customize `diff`-like comparators for their specific applications (e.g., they may ignore different timestamps). However, such tools must be manually configured for each project and potentially each test case, a human-intensive process that may need to be updated frequently as the test suite evolves. In following sections we will present a flexible technique for reducing false positives associated with a naive `diff` comparator which is able to learn the nature of faults in a target application rather than rely on human customization.

3 A Model of Test Case Output Differences

Our goal is to save effort in regression testing by using an automated test case output comparator. We specifically target applications with tree-structured output, such as XML or HTML, where a standard `diff` comparison would yield a high rate of false positives. The key distinction between tree-structured output and flat text files is that trees are well-formed objects with a directed edge relationship. In XML, for example, an element is defined between a start and end tag, which can contain other elements, attributes, and text.

Although recent work has explored using semantic graph differencing [20] and abstract syntax tree matching [6, 17] for analyzing source code evolution, such approaches are not helpful in comparing XML and HTML text outputs. First, because they depend on the presence of source code constructs such as functions and variables, which are not present in generic HTML or XML, to make distinctions. More importantly, however, they are meant to summarize changes, rather than to decide whether or not an update signals an error.

SMART relies on tree representations to make decisions about the relative importance of changes between two versions of output with respect to their structural significance.

We characterize each pair of program outputs via a distance metric that is based on the weighted sum of individual features. If the distance exceeds a given cutoff, the two test case outputs are deemed different enough to merit human inspection.

Source code differencers hard-wire knowledge of program semantics; SMART learns the weights and the cutoffs on a per-project or global basis via linear least-squares regression. An important advantage of linear regression over other techniques is that the resulting models are straightforward to analyze. We use analyses of variance to assess the relative contribution of our various features to an accurate prediction.

4 Comparing Pairs of Documents

Our approach classifies test case output based on structural and syntactic features of tree-structured documents. Although some are complicated, most features are quite simple, such as counting the number of inserted elements when converting one tree into the other. As a concrete example of a feature, consider counting node inversions: in HTML, `<b><u>text</u></b>` renders identically to `<u><b>text</b></u>`, even though the order of the bold and underline tags has been reversed.

4.1 Tree Alignment

To recognize such features, we must first align the input trees by matching up nodes with similar elements. An alignment is a partial mapping between the nodes of one tree and the nodes of the other. To see why this alignment is necessary, consider these two HTML fragments:

```
1  <u><b>textA</u></b> <i><u><b>textB</b></u></i>
2  <i><b><u>textB</u></b></i>
```

We must know how the fragments align before we can count inversions: if we align the `textB` subtree of #2 with the `textA` subtree of #1, we can count an inversion between the `<u>` and `<b>` tags. However, if we align the `textB` subtree of #2 with the `textB` subtree of #1, we can count inversions between the `<u>`, `<b>` and `<i>` tags.

This insight motivates us to find an alignment based on the minimal number of changes that describe the difference between two documents. We adapt the DIFFX [1] algorithm for calculating structural differences between XML documents to compute alignments on general tree-structured data. Matching pairs of elements between the newer and older trees allows SMART to identify local features derived from element pairs, as well as global features, such as the addition of natural language text across elements in the document.

Our technique employs features that fall into two loose categories: identifying differences in the higher level tree structure of the output, and emulating human judgment of interesting differences between two XML or HTML pages.

4.2 Tree-based differences

SMART uses features based on the tree structure of the test case output to signal interesting changes that merit human inspection. Taken together, these tree-based features are meant to flag a wide assortment of differences identified between two XML or HTML files, based on the tree structure itself. We arrived at several of our tree features through hours of manual inspection of test case output.

The DIFFX Algorithm. Three of these features are taken from a variant of the DIFFX [1] algorithm that we adapted to work on arbitrary tree-structured inputs rather than just XML. The algorithm computes the number of moves, inserts and deletes required to transform the first input into the second. It does this via bottom-up exact tree matching combined with top-down isolated tree fragment mapping; this amalgamated approach provides a high quality characterization of the relationship between the two input trees. We hypothesize that deletes and especially moves frequently correlate with test cases that merit inspection, and that the size of the change is indicative of the likelihood of an error. For example, if a generated webpage is missing a repeated element, or contains a long stack trace, the delete and insert features will allow our technique to flag it as worth inspecting. Considering moves instead of delete-insert pairs reduces the number of false positives returned.

Inversions. We hypothesize that inverted elements in XML or HTML do not indicate high-level semantic errors. We measure two related types of inversions. In both cases, we perform a pre-order traversal of all nodes in both of the document trees. Since we are interested in structural inversions, we remove all text nodes. We then sort the two lists and calculate the longest common subsequence between them. The longest common subsequence provides a specialized mapping between the two documents. We remove all nodes not in the common subsequence and unsort the lists, returning the remaining nodes to their original relative orders. We then compare the lists element-wise and count each difference; each difference is a structural inversion.

Grouped Changes. In addition to detecting changes to individual tree elements, we also detect when a set of elements that form a contiguous subtree are changed as a group. We measure the size of the grouped change in terms of the number of elements involved. We hypothesize that clustered edits are more likely to be deserve inspection, often because they contain missing components or lengthy exception reports. We also record a boolean feature that notes the presence or absence of grouped changes.

Depth of Changes. We note the relative depth of any edit operation within a tree. We hypothesize that changes

closer to the root are more likely to signal large semantic differences and thus more likely to merit human inspection.

Changes to Only Text Nodes. In many changes the differences between two outputs will be limited to text nodes while the tree structure remains unchanged. We expect that documents with such text-only differences are unlikely to contain semantic errors and thus should not be inspected.

Order of Children. We note when two aligned nodes are otherwise similar but have the order of their children changed. We hypothesize that changes in the order of children (as opposed to changes in the order of attributes) do not indicate high-level semantic errors and thus should not be inspected. This feature is complementary to moves being associated with errors in test case output.

4.3 Human-Judgment differences

We also attempt to detect changes a human would discern between two rendered versions of output files. These features are specific to HTML and attempt to identify differences that a human observer would notice on a web browser.

Text and Multimedia Ratios. Natural language and images play an important role in the human interpretation of a webpage. We measure the ratio of displayed text between two versions as well as the ratio of displayed text to multimedia objects. Replacing a small amount of text with an image, such as replacing a textual link with a button, is not a large semantic difference. On the other hand, changing many words in a small document may merit inspection.

Error Keywords. Many classes of errors in web applications follow similar patterns. For `NullPointerException`s and many other violations are tied to the underlying languages, and can be reasonably predicted by a textual search of the document for error keywords, such as "exception". Relying on natural language text to signal page errors has been previously explored in [2]. We hypothesize that output pairs containing error keywords in the newer version, but not in the older, are likely to merit human inspection.

Changes to Input Elements. We measure the addition and removal of functional elements of a webpage, such as buttons and forms. Beyond hyperlinks, these elements represent the primary interface between the user and the web application. We hypothesize that the removal of a functional element, such as a submit button, indicates that the output should be inspected.

Changed or Missing Attribute Values. We note when two aligned elements contain the same attribute but have a different attribute value. Consider this example:

```
1  < <Type id="_8" name="int"/>
2  ---
3  > <Type id="_8" name="unsigned int"/>
```

If the two `<Type>` elements on lines 1 and 3 are aligned then the change from `"int"` to `"unsigned int"` represents a meaningful change. Note that this is different than the case where the second `<Type>` has a new attribute that the original does not. Whether or not a change of an attribute value signals an error will depend on the specific application; the example above is more likely to indicate a bug than changing the value of an HTML image's `height` attribute. On the other hand, we claim that a newer version of code that is missing an attribute should be inspected.

5 Experimental Results

We evaluated SMART on ten open-source benchmarks that produced either XML or HTML output, totaling 473,000 lines of code. We chose benchmarks from an assortment of domains, considering only benchmarks for which multiple versions were available and for which a set of test cases was available. Figure 1 summarizes the programs used.

For each benchmark, we manually inspected the test case output generated by the two versions of the benchmark indicated. Our manual inspection marked the output as "definitely not a bug" or "possibly a bug, merits human inspection". We conservatively erred on the side of requiring human inspection, and annotated each test case twice, re-examining situations where the two annotations did not initially agree, so that our annotations remained consistent. Human inspection represents the standard software engineering processes and might, for example, discover problems in the program, the test case, or both. Our initial experiments involve 7154 pairs of test case output, of which 919 were labeled as requiring inspection.

For the hypothetical benefit experiment described in Section 5.3, we also considered multiple additional versions of two of the projects and manually annotated the test case output between each pair of successive versions. In sum, we annotated and evaluated on 20833 pairs of test case outputs.

5.1 Experiment 1 – Model Selection

In this experiment, we evaluate our technique's feasibility when phrased as an information retrieval task by creating a linear regression model based on those features and **selecting an optimal cutoff** to form a binary classifier.

5.1.1 Recall and Precision

We use *recall* and *precision* to evaluate the performance of our model. These measures are commonly used to evaluate document retrieval systems [25]; we use our model to answer the query: "Which test case outputs should be inspected by a human?" Precision and recall are defined as

Benchmark	Versions	LOC	Description	Test cases	Test cases to Inspect
HTMLTIDY	Jul'05 Oct'05	38K	W3C HTML validation	2402	25
LIBXML2	v2.3.5 v2.3.10	84K	XML parser	441	0
GCC-XML	Nov'05 Nov'07	20K	XML output for GCC	4111	875
CODE2WEB	v1.0 v1.1	23K	pretty printer	3	3
DOCBOOK	v1.72 v1.74	182K	document creation	7	5
FREEMARKER	v2.3.11 v2.3.13	69K	template engine	42	1
JSPPP	v0.5a v0.5.1a	10K	pretty printer	25	0
TEXT2HTML	v2.23 v2.51	6K	text converter	23	6
TXT2TAGS	v2.3 v2.4	26K	text converter	94	4
UMT	v0.8 v0.98	15K	UML transformations	6	0
Total		473K		7154	919

Figure 1. The benchmarks used in our experiments. The "Test cases" column gives the number of regression tests we used for that project; the "Test cases to Inspect" column gives the number of those tests for which our manual inspection indicated a possible bug.

follows:

$$\text{recall} = |D \cap M| \div |D| \qquad \text{precision} = |D \cap M| \div |M|$$

where D is the *desired* set of test cases (as determined by our human annotators, see Section 5); and M (for *model*) is the set of test cases returned by our technique.

Precision can be trivially maximized by returning a single test case, while *recall* can similarly be maximized by returning all test cases. We avoid these scenarios by combining the two measures by taking their harmonic mean. The resulting F_1*-score* is defined as $F_1 = 2pr \div (p + r)$, where p is precision and r is recall. This metric gives equal weight to precision and recall. In practice *recall*, which penalizes missing real bugs, is usually more important; we will return to that consideration in Section 5.3.

5.1.2 Cross Validation

Before evaluating the performance of our model, we must first ensure that it is not biased with respect to our data set. To mitigate the threat of overfitting, we used 10-fold cross validation [13]. Although cross-validation may not always be necessary for linear regression, we performed it explicitly to ensure our results were not biased by testing and training on the same data. We randomly assigned test cases from all programs in the data set into ten equally-sized groups. Each group is reserved once for testing, and the remaining nine groups are used to train the model; we thus never train and test on the same data. Next we average the cross validation results and compare these values to the results of the same model when we trained and tested on the entire data set. If the two outcomes were not significantly different, we can conclude that we were not subject to bias.

5.1.3 Experimental Procedure

SMART classifies pairs of tree-structured outputs based on whether a human should inspect them or not. In this experiment:

1. We first perform the cross-validation steps (Section 5.1.2). On each fold, we train a linear model as if the response variable (i.e., our boolean human annotation of whether a human should inspect that output or not) were continuous in the range [0,1].

2. The real-valued model outputs are turned into a binary classifier by comparing against a cutoff. We perform a linear search to find a model cutoff. Depending on how the result of applying the linear model compares to the cutoff, SMART reports that the outputs need be or need not be inspected. We choose the cutoff and comparison that yield the highest F_1-score for each validation step.

3. After cross-validation, we train the model on the entire data set. We again find the best model cutoff and comparison to maximize the F_1-score.

5.1.4 Results

Figure 2 shows our precision, recall, and F_1-score values for our dataset. As a point of comparison, we also computed the predictive power of `diff`, `xmldiff` [34], coin toss, and biased coin toss as baseline values. The fair coin returns "no" with even probability. The biased coin returns "no" with probability equal to the actual underlying distribution for this dataset: $(7154-919)/7154$. Note that the biased coin toss cannot generally be implemented in the field since it relies on knowing the distribution of right answers in advance. Despite this, SMART has clear advantages in predictive power over `diff`, `xmldiff`, and random or biased chance; our approach yields three times `diff`'s F_1-

Comparator	F_1-score	Precision	Recall
SMART	0.9931	0.9972	0.9890
SMART w/ cross-validation	0.9935	0.9951	0.9920
`diff`	0.3004	0.1767	1.0000
`xmldiff`	0.2406	0.1368	1.0000
fair coin toss	0.2045	0.1286	0.4984
biased coin toss	0.2268	0.1300	0.8868

Figure 2. The F_1-score, precision, and recall values for SMART on our entire dataset. Results for `diff`, `xmldiff`, and random approaches are given as baselines; `diff` represents current industrial practice.

score. `xmldiff`, an off-the-shelf `diff`-like tool for XML and HTML [34], was a worse comparator than than basic `diff` because it was unable to process some benignly ill-formatted output.

Little to no bias was revealed by cross-validation. The absolute difference in F_1-score between the model and its corresponding averaged cross validation steps was 0.0004. This shows that results obtained using the corresponding model trained on the entire data set were never significantly different from the the averaged results from each set of cross validation steps.

After measuring our performance for our global model, we evaluated our performance when training on each project individually. We found both that we can train an effective model using only information from one project, and also that we can learn a *more* effective model using per-project information. In addition, we found that the effects of our features were frequently similar across different projects.

5.2 Experiment 2 – Feature Analysis

In this section, we **evaluate relative feature importance** and determine which features correlate with output that should be inspected. Figure 3 shows the results of a per-feature analysis of variance on the model using the entire dataset. The table lists only those features with a significant main effect. F denotes the F-ratio, which is close to 1 if the feature does not affect the model; conceptually F represents the square root of variance explained by that feature over variance not explained. The p column denotes the significance level of F (i.e., the probability that the feature does not affect the model).

5.2.1 Results

Our most significant feature was whether or not the change involved only low-level text. This is the key distinction between our approach for tree-structured output and the state-of-the-art for normal textual output: in normal practice, changes to the text of the output indicate regression errors. We find, however, that text-only changes have a strong negative effect: test case outputs that differ only in small amounts of non-structural text do *not* merit human inspection. This is one of the key reasons we are able to outperform `diff`, because many web-based applications may update natural language text as part of normal program evolutions.

Feature	Coefficient	F	p
Text Only	- 0.217	179000	< 0.001
DIFFX-move	+ 0.003	170000	< 0.001
DIFFX-delete	+ 0.017	52700	< 0.001
Grouped Boolean	+ 0.792	9070	< 0.001
DIFFX-insert	+ 0.019	862	< 0.001
Error Keywords	+ 0.510	410	< 0.001
Input Elements	+ 0.118	184	< 0.001
Depth	- 0.001	128	< 0.001
Missing Attribute	- 0.045	116	< 0.001
Children Order	- 0.000	77	< 0.001
Grouped Change	- 0.078	62	< 0.001
Text/Multimedia	+ 0.009	19	< 0.001
Inversions	- 0.000	6	0.020
Text Ratios	- 0.001	6	0.020

Figure 3. Analysis of variance of our model. A + in the 'Coefficient' column means high values of that feature correlate with test cases outputs that should be inspected (both + and - indicate useful features). The higher the value in the 'F' column, the more the feature affects the model. The 'p' column gives the significance level of F; features with no significant main effect ($p \geq 0.05$) are not shown.

Our DIFFX-move feature was frequently correlated with test case errors. It may seem counter-intuitive that moves, as opposed to insertions or deletions, would indicate a need for human inspection. In practice, however, tree-structured moves show up as a side-effect of other large changes; the introduction or deletion of one element often involves a move of its neighbors. Despite the high F-ratio of the DIFFX-move feature, its model coefficient was an order of magnitude smaller than those of insert or delete. Although moves were most frequently associated with errors, other features also had to be present in order for the test case output to merit inspection.

Our boolean feature that indicated the presence or absence of clustered changes was also highly correlated with errors. We claim that variations in the sizes of the grouped changes are not as salient as their existence. We note that grouped changes were more important than DIFFX-inserts, which may have been scattered across the output.

Some of our features were less powerful than we origi-

nally hypothesized. For example, the presence of error keywords did not effect our model as much as the features listed above. Analysis of natural language to identify errors can be challenging [11], and will likely vary between different data sets. The high coefficient for error keywords allows them to overcome the negative impact of text-only changes, because error keywords only occur in text.

5.3 Experiment 3 – Effort Saved

Finally, we **evaluate the hypothetical benefit** of our technique, in terms of developer effort saved, when used to determine if humans should inspect regression test output over multiple revisions to software projects. We consider a situation in which a development organization uses our technique on all regression tests between successive releases of the same project. We assume that humans manually inspect a small percentage of the test case output flagged by `diff` — 20% in this experiment — and then train our tool on that information (as in Section 5.1.3), using it to guide the inspection of the remaining test cases. Subsequent releases of the same project retain training information from previous releases, as well as incorporating the false positive or true positive results of any test case that our tool deemed to require manual inspection.

5.3.1 Experimental Procedure

Two of our benchmarks, GCC-XML and HTMLTIDY, had three or more released versions available. For each successive version considered, we manually annotated the test case differences to determine if a human should inspect them (see Section 5). The dataset for this simulation thus includes 20232 regression test output pairs spanning seven software releases for two projects. Note that this is slightly different setup than that of Figure 1 — for example, while there were 25 test cases to inspect for the version of HTMLTIDY used in Figure 2, here we use five different releases of HTMLTIDY that have correspondingly different numbers of test outputs that should be inspected (12–254).

We measure the number of test cases flagged for manual inspection (i.e., the 20% used for initial training as well as the true positives and false positives produced by our tool) as well as the number of false negative test cases that should have been flagged for inspection (i.e., that indicated potential bugs found via regression testing) but were not. Each of these carries an associated software engineering cost.

We can estimate the amount of effort saved by developers when using SMART, by defining a cost of looking (*LookCost*) at a test case and a cost of missing (*MissCost*) for each test case that should have been flagged but was not. We consider SMART a useful investment when the cost of using it:

$$(TruePos + FalsePos) \times LookCost + FalseNeg \times MissCost$$

is less than the cost of $|\texttt{diff}| \times LookCost$. That is, SMART saves effort when the cost of looking at the test cases flagged by `diff` but not by our technique exceeds the cost of missing any relevant test cases we fail to report. We thus express the condition under which our technique is profitable:

$$\frac{LookCost}{MissCost} > \frac{-\ FalseNeg}{TruePos + FalsePos - |\texttt{diff}|}$$

We assume *LookCost* $\ll$ *MissCost*, so we would like this ratio to be as small as possible.

5.3.2 Results

Figure 4 shows our results. For example, when applying our technique to the last release of HTMLTIDY, the ratio above which we are profitable is about 1/1000; if the cost of missing a potentially useful regression test report is less than or equal to 1000 times the cost of triaging and inspecting a test case, we save developer effort. A ratio of 0 indicates that we have no false negatives, and in such cases we always outperform `diff`, regardless of *LookCost* or *MissCost*. Figure 4 also shows the number of test cases that a developer would need to examine when using `diff`.

SMART's performance generally improves on subsequent releases, and it totally avoids false negatives in one instance for both benchmarks. Our model is at its worst when there is a large relative number of regression test errors (e.g., for a rushed release that fails to retain required functionality). For the fourth release of HTMLTIDY, the number of test cases that should be inspected is a order-of-magnitude higher than usual. A cautious development organization might use our tool only when the random sampling of 20% of the test case outputs shows a historically reasonable number of regression test errors.

Previous work on bug report triage has used a *LookCost* to *MissCost* ratio of 0.023 as a metric for success for an analysis that required 30 days to operate [10], and we adopt that ratio as a baseline here. The typical performance of our technique, which includes the cost of the 20% manual annotation burden and would take 1.3 hours on average per release, is 0.0183 — a 20% improvement over that figure. If we exclude the HTMLTIDY outlier mentioned above our ratio is 0.0015; we exceed the utility of previous tools by an order of magnitude and require an order of magnitude less time.

In general, *LookCost*, the time to compare the results of a test suite to the oracle, is typically a few minutes for each test case [18]. *MissCost* varies by application domain: it can be low for applications where servers can easily update deployed software, but is often high for web applications with high quality-of-service requirements [37]. An IBM publication provides concrete examples of industrial values for these costs: based on a 2008 report [36], *LookCost* is

Benchmark	Release	Test Cases	Should Inspect	True Positive		False Positives		False Negatives		Ratio
				SMART	diff	SMART	diff	SMART	diff	
HTMLTIDY	2nd	2402	12	5	12	78	781	7	0	0.0099
	3rd	2402	48	48	48	0	782	0	0	0
	4th	2402	254	109	254	1	574	145	0	0.2019
	5th	2402	48	48	48	0	775	0	0	0
	6th	2402	20	19	20	1	774	1	0	0.0013
GCC-XML	2nd	4111	662	658	662	16	2258	4	0	0.0018
	3rd	4111	544	544	544	0	2577	0	0	0
total		20232	1588	1431	1588	96	8521	157	0	0.0183

Figure 4. Simulated performance of our technique on 20232 test cases from multiple releases of two projects. The 'Test Cases' column gives the total number of regression tests per release. The 'Should Inspect' column counts the number of those tests that our manual annotation indicated should be inspected (i.e., might indicate a bug). The 'Inspected' column gives the number of tests that our technique and `diff` flag for inspection. The 'False Positives' and 'False Negatives' columns measure accuracy, and the 'Ratio' column indicates the value of *LookCost*/*MissCost* above which our technique becomes profitable (lower values are better).

$25 and *MissCost* is $450 (the cost of a defect "during the QA/testing phase"). With those cost figures, using SMART reduces the costs associated with regression testing over all releases shown in Figure 4 by 48% ($131730 vs. $252725). Even if *MissCost* doubles to $1000, SMART still reduces the costs by 22% ($195175 vs. $252725).

5.4 Threats to Validity

Although we outperform `diff` by over a factor of three, it is possible that our results do not generalize to industry practice for various reasons. For example, the benchmarks used in our experiments may not be representative of other projects. To mitigate this threat, we selected our two large benchmarks (HTMLTIDY and GCC-XML) from different domains, and supplemented our global dataset with other, smaller benchmarks to increase the diversity of the data we test on. It is possible that some results are more indicative for our two larger benchmarks than the smaller ones, however, and in future work we would like to include more test applications. Even if our benchmarks are representative, it is possible we overfit our model to the data; our cross-validation in Section 5.1.2 suggests that is not the case.

Similarly, our results may not generalize if we do not use SMART as developers would in practice: we examined the regression test output of versions of HTMLTIDY and GCC-XML that were several months apart, but in practice some organizations may perform these tests more frequently, such as during a nightly build. Because our model's performance depends on the relative frequency of bugs between release versions, and not the number of bugs in general, we believe that we will still be able to save developer effort, even if regression tests are run more frequently.

Finally, our human annotations may not have accurately flagged potential errors in regression test output. To avoid missing actual bugs, our annotations were conservative: we only annotated test outputs as not meriting inspection if we were highly certain they did *not* indicate an error. Thus we may have annotated non-errors as errors, which translates into less of an opportunity for us to outperform `diff`, but does not impact the correctness of our approach. In addtion, because annotators were also responsible for suggesting some features for the model, it is possible that bias exists in the annotations themselves, although care was taken to avoid this situation and baseline annotations were computed at least twice on each output pair to guarantee a minimum level of consistency.

5.5 Experimental Summary

In this section we have shown that we can use our syntactic and structural features to build a model that classifies which regression test case outputs merit human inspection. On 7154 test cases from 10 projects, SMART obtains a precision of 0.9972, a recall of 0.9890, and an F_1-score of 0.9931, over three times as good as `diff`'s F_1-score of 0.3004. Although these are very strong machine learning results, we further tested our technique in a simulated deployment involving 20232 test cases and multiple releases of two projects. In that scenario we had 8425 fewer false positives than `diff`, and we save development effort when the ratio of the cost of inspecting a test case to the cost of missing a relevant report is over 0.0183; we claim that numbers in that range correspond to a savings for typical industrial practice.

6 Related Work

Sprenkle *et al.* have focused on oracle comparators for testing web applications [29, 30, 31] with HTML output. Building on a capture-replay testing framework for user session data [29], they investigate features based on `diff`, content, and structure. They refine these features into oracle comparators [31] based on HTML tags, unordered links, tag names, attributes, forms, the document, and content. They then investigate applying decision tree learning to identify the best combination of oracle comparators for specific applications [30]. We also combine machine learning and automated oracle comparators, but we include features and experiments that are not HTML-specific and can be applied to any tree-structured data. Finally, they validate their approach by measuring their oracles' abilities to reveal seeded defects in a single version of an application (i.e., measuring differences between the clean application and a fault-seeded one). By contrast, our experiments train and test on data between *different versions* of the same application. Our approach aims to not flag common and benign program evolutions, in contrast to the setting of Sprenkle *et al.* [30], where a deterministic application would yield no false positives with a `diff` comparator.

Lucca et al. address the open issues of web application testing through the use of an object-oriented model of a web application which defines the unit level for testing [15]. As part of their testing toolkit, they outline a *Comparator* which automatically compares the actual results against the expected values of the test execution. The authors do not describe the details of their comparator, but SMART can be thought of as a working instantiation of such a design.

Many testing methodologies use oracle comparators that require manual intervention in the presence of discrepancies [5, 15, 22, 31]. For example, user session data can be used as both input and also test cases [5, 29], but there must be a way to compare obtained results with expected results. Lucca *et al.* address web application testing with an object-oriented web application model [15]. They outline a comparator which automatically compares the actual results against the expected values of the test execution. Our technique can be thought of as a working instantiation of such a design, and we extend the notion to structural differences.

Sneed explores a case study on testing a web application system for the Austrian Chamber of Commerce [27]. A capture-replay tool was used to record the dialog tests, and XML documents produced by the server were compared at the element level: if the elements did not match, the test failed. Our approach also compares XML documents, but does not necessarily rely on exact element matching, and thus reports fewer false positives.

Haran *et al.* aim to automatically support fault detection by classifying program executions [7]. They also use machine learning to build a model of program executions that are likely to be associated with failed test cases. Their approach is orthogonal to ours, although it may be difficult to apply their approach to some web applications due to the dynamic nature of content generation.

7 Conclusions

We present SMART, a technique for reducing the cost of regression testing by using syntactic and structural features to decide whether or not test case output merits human inspection. In domains with tree-structured output, such as HTML, XML or abstract syntax trees, traditional `diff`-based comparisons yield too many false alarms. We posit a number of features that can be used to distinguish potential errors from harmless functionality additions or rendering changes. For example, changes to the text of HTML output are actually negatively correlated with the presence of potential errors, while tree-structured differences, such as moving a subtree from one part of the output to another, are positively correlated with potential errors.

We evaluate SMART both as a model and as a cost-saving technique. As a model evaluated on 7154 test case pairs from 10 projects, we obtain a precision of 0.9972, a recall of 0.9890 and an F_1-score of 0.9931, over three times as good as the standard `diff` F_1-score of 0.3004. We complement these strong machine learning results with a simulated deployment involving 20232 test cases. SMART had only 96 false positives — 8425 fewer than `diff` — and saves development effort when the ratio of the cost of inspecting a test case to the cost of missing a relevant report is over 0.0183, a range both corresponding to a savings for typical industrial practice and also 20% better than previously-published results. Web applications and XML-processing middleware applications are becoming increasingly important; SMART presents a first step toward a comparator that makes regression testing for them attractive.

References

[1] R. Al-Ekram, A. Adma, and O. Baysal. diffX: an algorithm to detect changes in multi-version XML documents. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

[2] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *World Wide Web Conference*, May 2002.

[3] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[4] B. Boehm and V. Basili. Software defect reduction. *IEEE Computer Innovative Technology for Computer Professions*, 34(1):135–137, January 2001.

[5] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *International Conference on Software Engineering*, pages 49–59, 2003.

[6] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[7] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.

[8] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[9] E. Hieatt and R. Mee. Going faster: Testing the web application. *IEEE Software*, 19(2):60–65, 2002.

[10] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.

[11] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks*, pages 52–61, 2008.

[12] G. M. Kapfhammer. Software testing. In *The Computer Science Handbook*, chapter 105, 2004. CRC Press.

[13] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[14] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming language design and implementation*, pages 141–154, 2003.

[15] G. D. Lucca, A. Fasolino, F. Faralli, and U. de Carlini. Testing web applications. *International Conference on Software Maintenance*, page 310, 2002.

[16] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. DART: A framework for regression testing "nightly/daily builds" of GUI applications. In *International Conference on Software Maintenance*, 2003.

[17] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[18] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Commun. ACM*, 41(5):81–86, 1998.

[19] S. Pertet and P. Narsimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, Dec. 2005.

[20] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. pages 188–197, 2004.

[21] C. V. Ramamoothy and W.-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[22] F. Ricca and P. Tonella. Testing processes of web applications. *Ann. Softw. Eng.*, 14(1-4):93–114, 2002.

[23] S. Rosenberg. What you don't know can cost you millions, July 2009. `http://www.cxoamerica.com/pastissue/article.asp?art=270091&issue=202`.

[24] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[25] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[26] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[27] H. M. Sneed. Testing a web application. In *Workshop on Web Site Evolution*, pages 3–10, 2004.

[28] F. Soliman and M. A. Youssef. Internet-based e-commerce and its impact on manufacturing and business operations. In *Industrial Management & Data Systems*, pages 546–552. MCB UP Ltd, 2003.

[29] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Automated Software Engineering*, pages 253–262, 2005.

[30] S. Sprenkle, E. Hill, and L. Pollock. Learning effective oracle comparator combinations for web applications. In *International Conference on Quality Software*, pages 372–379, 2007.

[31] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *International Symposium on Reliability Engineering*, pages 117–126, 2007.

[32] J. Sutherland. Business objects in corporate information systems. *ACM Comput. Surv.*, 27(2):274–276, 1995.

[33] `http://txt2html.sourceforge.net/`. txt2html - text to HTML converter. Technical report, 2008.

[34] `http://www.a7soft.com/jexamxml.html`. JExamXML diff tool for comparing and merging xml documents. Technical report, 2008.

[35] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Timeaware test suite prioritization. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

[36] L. Williamson. IBM Rational software analyzer: Beyond source code. In *Rational Software Developer Conference*. `http://www-07.ibm.com/in/events/rsdc2008/presentation2.html`, June 2008.

[37] Y. Wu, D. Pan, and M.-H. Chen. Techniques of maintaining evolving component-based software. *International Conference on Software Maintenance*, page 236, 2000.

[38] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen. Regression testing for web applications based on slicing. In *Conference on Computer Software and Applications*, pages 652–656, 2003.