Learning for Test Prioritization: An Industrial Case Study

Benjamin Busjaeger
Salesforce.com
San Francisco, CA 94105
bbusjaeger@salesforce.com

Tao Xie
University of Illinois at Urbana-Champaign
Urbana, IL 61801
taoxie@illinois.edu

ABSTRACT

Modern cloud-software providers, such as Salesforce.com, increasingly adopt large-scale continuous integration environments. In such environments, assuring high developer productivity is strongly dependent on conducting testing efficiently and effectively. Specifically, to shorten feedback cycles, test prioritization is popularly used as an optimization mechanism for ranking tests to run by their likelihood of revealing failures. To apply test prioritization in industrial environments, we present a novel approach (tailored for practical applicability) that integrates multiple existing techniques via a systematic framework of machine learning to rank. Our initial empirical evaluation on a large real-world dataset from Salesforce.com shows that our approach significantly outperforms existing individual techniques.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Regression testing, test prioritization, learning to rank

1. INTRODUCTION

Regression testing is a crucial activity in the software development process to ensure that changes do not adversely affect existing functionality. However, it is costly to conduct regression testing in terms of both time and resources. To reduce cost, various regression test optimization mechanisms, such as test selection and prioritization, have been developed. Test selection aims to identify the exact set of tests (affected by the changes) that have to be run. In practice, test selection often either underestimates or overestimates the set of tests needed [6]. Test prioritization [11] orders tests to maximize the likelihood of meeting a certain objective, typically revealing failures earlier. Test prioritization can also be used for test selection in resource-constrained environments by running the top-k ranked tests [6].

Test optimization mechanisms are particularly important for modern cloud-software companies, such as Salesforce.com, given the high change frequency and massive scale of the companies' code bases. Engineers typically continuously integrate their work into the mainline from which releases are deployed directly into production [5, 16]. Code health of the main branch is vital and small improvements in efficiency yield high gains in productivity. In particular, test prioritization can help in two main ways. First, test prioritization can be applied before submitting code to run a subset of tests most likely to fail. Doing so gives engineers fast feedback before switching context and can prevent major regressions from entering the mainline. Second, test prioritization can be applied after submitting code to run tests in the order of fault-revealing likelihood. Even with modern large-scale parallel test infrastructures, it can take a relatively long time until an engineer has received complete testing feedback for a given change. Test prioritization paired with automated bug assignment can shorten the feedback window to ensure that issues having entered the code base are resolved in a timely manner.

Transferring academic research on test prioritization [11] into an industry setting requires significant adaptation to account for practical realities of heterogeneity, scale, and cost. Various existing techniques and studies on test prioritization [11] focus on relatively small, single-language, unit-tested projects. On the other hand, the target industrial system under test in this research is a large software system, written in multiple languages, with extensive integration-test suites. Some requirements of existing techniques, such as availability of per-test code coverage for every change, is difficult to attain in such environments [5]. Furthermore, the use of non-code artifacts, such as configurations, is typically not accounted for by coverage-based techniques [9].

To address such challenges in industrial environments, we present a novel approach for test prioritization focused on practical applicability. To cope with heterogeneity, we apply multiple heuristic techniques shown by existing research to perform well individually: test coverage of modified code [11], textual similarity between tests and changes [12], recent test-failure or fault history [6], and test age [5]. Each of these heuristic techniques excels for a certain type of changes and tests. For example, code coverage [11] can identify complex interdependencies between seemingly unrelated parts of the system under test, such as impacts of low-level persistence-logic changes on user-interface tests. Text sim-

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2983954

ilarity [12] performs particularly well for non-code changes. For example, configuration-file changes not captured by coverage metrics often contain similar terms to those from impacted tests. Fault history [6] accounts for temporal relationships, which help identify tests impacted by churning or non-deterministic code; these tests would otherwise be ranked lower. Finally, boosting new tests [5] alleviates the cold-start problem of having no prior coverage, text, or fault metrics. To handle the large scale at reasonable cost, we collect data asynchronously and accommodate potential inaccuracies in these heuristic techniques. For example, coverage scores are computed based on proximity of covered and changed lines rather than exact statement or block coverage.

Applying multiple techniques together introduces the challenge of how to integrate them effectively. It is not immediately evident how individual scores produced by these different techniques should be combined into an aggregate score by which tests can be ranked. Furthermore, how scores should be aggregated may depend on the types of changes under consideration. For example, intuitively, code-coverage-based scores should be emphasized for changes that affect mostly source code. Rather than devising a model for ranking tests through tedious and error-prone manual experimentation, we draw on foundational results of learning to rank from the field of information retrieval (IR) and machine learning [7]. In particular, we use past test results, along with information about each change that induced test failures and successes, to systematically train a ranking model. The trained model is then used to predict rankings for unseen tests based on properties of each change and each test.

The main practical benefits of our approach are that it leverages abundant data of test results available in practice, facilitates exploration of new predictive features, and performs well for various types of changes and tests. To the best of our knowledge, our work is the first that reduces the problem of test prioritization to that of learning to rank. Our initial empirical evaluation on a large real-world dataset from Salesforce.com shows that our approach substantially outperforms existing individual techniques. This paper makes the following main contributions:

- Discussion of main challenges encountered and insights gained in conducting test prioritization in an industrial setting.
- A novel approach of test prioritization that systematically integrates existing individual techniques by learning from past test results.
- An empirical evaluation of our approach using a large real-world dataset from Salesforce.com.

2. BACKGROUND AND RELATED WORK

Early techniques of test prioritization [11] focused mostly on evaluating various code coverage metrics as the ranking criteria. Independent variables include whether all statements (total) or only previously not-covered statements (additional) are measured [11], whether or not code changes are considered [3], and the granularity of coverage data (class, function, statement, etc.). In general, additional, change-aware, fine-grained coverage yields higher accuracy, but differences are often insignificant [3]. We adopt a total, change-aware coverage heuristic that scores based on proximity between changes and executed code to accommodate slightly out-dated coverage information. We also apply the Average Percentage of Faults Detected (APFD) metric defined by Rothermel et al. [11] in our evaluation: (n: number of tests, m: number of failures, TF_i: rank of ith failure)

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{1}{2n} \quad (1)$$

History-based techniques rank tests based on past execution results. Kim and Porter [6] proposed the first technique that computes ranking scores using a smoothed, weighted moving average of past failures. Elbaum et al. [5] applied a similar windowing technique tailored to continuous integration environments. A more recent study [10] factored similarity between tests (in terms of coverage) into the historical weighting. We integrate a simple weighted average of recent test failures into our model.

Recently, IR techniques have been successfully applied to rank tests. Saha et al. [12] indexed text content of tests and queried using words extracted from change diffs. Our approach includes a number of text-based features using similar techniques of document similarity.

There have also been industrial case studies on test prioritization. Microsoft applied test prioritization for testing Windows [13, 2] and Dynamics Ax [1]. Google evaluated prioritization to optimize pre- and post-submit testing for a large, frequently-changing code base [5, 16]. This setting is similar to ours. Cisco investigated prioritization for reducing long-running video conferencing tests [8].

Learning to rank [7] is a subfield of machine learning concerned with the construction of ranking models for IR. Training data consists of item lists with associated relevance judgments (usually either binary or ordinal), which induce a (partial) order over the items. The objective is to train models that rank unseen lists such that some ranking metric is maximized. Learning algorithms are categorized into point-wise, pair-wise, and list-wise. Point-wise algorithms reduce ranking to standard classification techniques, pair-wise algorithms classify pairs of points with the objective of minimizing inversions, and list-wise algorithms consider complete item lists using direct continuous approximations for discrete ranking metrics. Pair-wise and list-wise algorithms are more natural fits for learning to rank as they optimize an easier problem of predicting relative as opposed to absolute scores. Empirically, they outperform point-wise algorithms [7].

Important metrics include precision (the percentage of relevant items among all retrieved items) and recall (the percentage of relevant items retrieved). For most IR tasks, precision is more important, because users tend to look at only top-ranked items. For priority-based test selection, recall is more important, since the objective is to reveal as many faults as possible. Average precision (AP) is a commonly used metric in the presence of binary judgments for evaluating rankings. It averages precision across all recall points. This metric differs from APFD in that this metric does not take the total number of items into account, but provides insight on how many non-relevant items are ranked in front of relevant ones on average: (M: ranks of relevant items, R_k: ranking up to position k)

$$AP(Q) = \frac{1}{|M|} \sum_{k \in M} Precision(R_k) \quad (2)$$

3. TESTING AT SALESFORCE.COM

Salesforce.com is a multi-tenant cloud application and development platform [15]. Similar to other modern cloud-

software providers, most of the core code is continuously developed, integrated, and released out of a single large code base. The source repository contains many hundreds of thousands of files of various types. While Java is the most common language, many other languages are used, such as PL/SQL for optimized database access, JavaScript for user interface programming, and Apex [15] for higher-level business logic. In addition, large portions of the source artifacts are configuration files in the form of XML, properties, or schema definitions. The metadata-driven architecture of the application also involves extensive code generation and dynamic tests (e.g., parameterized tests whose parameters are derived from metadata).

The process of regression testing consists of three main phases supported by three types of continuous integration (CI) jobs (precheckin, producer, and consumer), respectively. First, changes are submitted to *precheckin* jobs, which build binaries against the latest source code, run a small suite of smoke tests, and commit into the main branch if all tests pass. Second, *producer* jobs deploy artifacts for the committed changes. Third, *consumer* jobs run various tests against the deployed artifacts. These tests are categorized by type, such as unit, functional, and user interface, and by bucket, such as basic, extended, and full. The latter serves as a manual prioritization mechanism. Each bucket runs periodically for batches of changes where batch size depends on job duration. To scale for large change and test volume, the test infrastructure heavily parallelizes test execution. At any given time, thousands of virtual machines run tests on multiple cores. Despite the massive infrastructure, it can take a relatively long time for engineers to get feedback, especially for extended buckets of functional or user interface tests.

A critical component in the testing system is the automated bugging service, which is responsible for identifying which change in a given batch introduced each previously unobserved test failure. In the majority of cases, this service simply reruns failing tests on changes in the batch using an optimized search procedure. While this technique is generally reliable, certain situations can lead to invalid assignments. Some examples include missing binary artifacts for changes due to broken builds or network/server issues, non-deterministic test failures (flaky/flapping tests), or severe faults that mask others. We obtained the training and test data for our evaluation from the database of this service.

The heterogeneity and scale of the system present three main challenges for conducting test prioritization. First, the rapid change/test frequency makes it infeasible to collect up-to-date static or dynamic program traces [5]. Specifically, collecting per-test Java code coverage is challenging, because existing tools require single-threaded execution. While it is possible to modify such tools to maintain thread-local state, doing so would impose untenable threading restrictions on tests. We also observed performance and behavior side effects from code instrumentation for collecting coverage information. Second, the code-base size is much larger than most projects studied in previous work, leading us to favor higher efficiency over accuracy for heuristics. Third, codebase diversity in terms of artifact types makes a single surrogate objective function based on coverage impractical. To account for heterogeneity, our objective function considers multiple facets simultaneously.

The two main use cases of test prioritization for the system under test are (1) running top k tests most likely to fail in precheckin jobs (to keep the code line healthy), and (2) running all tests in priority order in consumer jobs (to provide faster test result feedback).

4. APPROACH

In this section, we formally reduce the problem of test prioritization to that of learning to rank, and describe the features used for learning and the model being learned.

4.1 Problem Formulation

The following problem formulation for test prioritization is a straightforward transliteration of ranking creation as defined by Li [7]. There are two sets: the set of changes $\Delta = \{\delta_1, \delta_2, \ldots, \delta_M\}$ and the set of tests $\mathcal{T} = \{t_1, t_2, \ldots, t_N\}$. In IR, these sets correspond to queries and documents, respectively. Given a change $\delta \in \Delta$ and a subset $T \subseteq \mathcal{T}$, the task is to rank elements of T using information about δ and T.

Ranking is performed by sorting T according to scores s_T obtained from a scoring function $f(\delta, t) : \Delta \times \mathcal{T} \rightarrow \mathbb{R}$. A given ranking list is denoted as $\pi = sort_{s_T, t \in T}(T)$. Scores indicate the likelihood that a given test t will fail for a given change δ.

In the training data, relevance labels are associated with change/test pairs to induce a partial order among tests for a given change. We use binary relevance labels $\mathcal{Y} = \{0, 1\}$ to indicate whether a test passed or failed, respectively. Concretely, given a set of changes $\{\delta_1, \delta_2, \ldots, \delta_m\}$ along with, for each change δ_i, a set of tests $T_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,n_i}\}$ and associated labels $\mathbf{y_i} = \{y_{i,1}, y_{i,2} \ldots, y_{i,n_i}\}$ (where n_i is the number of tests observed for change δ_i), the training set is denoted as $S = \{(\delta_i, T_i), \mathbf{y_i}\}_{i=1}^{m}$. To facilitate learning, a feature vector $x_{i,j} = \phi(\delta_i, t_{i,j})$ is created for each change/test pair, where ϕ denotes the feature function. Therefore, letting $\mathbf{x_i} = \{x_{i,1}, \ldots, x_{i,n_i}\}$, the training set can be written as $S' = \{(x_i, y_i)\}_{i=1}^{m}$.

The learning task is to infer a ranking model in the form of a scoring function $f(\delta, t) = f(x)$ that produces ranking lists (permutations) to maximize some ordering-based objective function for unseen data. Note that with binary labels, there are usually multiple optimal permutations.

4.2 Features

Our ranking model currently uses five features: Java code coverage, text path similarity, text content similarity, failure history, and test age. While these features are not an exhaustive set, they represent some of the most relevant indicators studied in previous work. We plan to study additional features such as file types and coverage for other languages in future work. A previous technique [14] on fault prediction used word frequency as features instead of computing text similarity scores. While this technique allows for more flexible and direct learning of correlation and fault proneness, it introduces additional practical challenges of having to train more frequently and account for concept drift. We plan to explore this trade-off in future work, including evaluating newer word vector models.

The first feature of our model is a per-test *code coverage* score. We use the open source Java Code Coverage Library (JaCoCo)[1] to instrument Java bytecode at runtime. Practical benefits of the library include obviating the need to rebuild source code and low induced performance overhead. We obtain per-test coverage by dumping collected coverage

[1]http://jacoco.org

information between test executions. Since this technique requires single-threaded execution, coverage-collection jobs currently run periodically in parallel to the actual testing jobs. We also have to apply an optimization to retain runtime performance[2] and modify some reflective application code to ignore bytecode instrumented for coverage collection. Previous research [4] has shown that coverage information quickly becomes outdated, so our heuristic does not entail precise coverage data. To enable efficient computation of coverage score, we build an inverted index that maps each source file to the line numbers executed by each test.

The coverage score for a given test is the number of changed lines executed by the test. Formally, let F_c be the set of files modified by change c, F_t be the set of files executed by test t, and $\mathcal{L}(t, f)$ be the mapping function that returns the number of lines executed by test t in file f. Then the coverage feature score ϕ_1 is computed as follows:

$$\phi_1(c,t) = \sum_{f \in F_c \cup F_t} \mathcal{L}(t,f)$$

The second and third features of our model are *text similarity* scores for test-file path and content, respectively. These features are similar to URL and page-body indexing in traditional IR systems. On these two features, our technique resembles Saha et al.'s technique [12]; however, our technique uses a slightly different similarity function and learns coefficients from data. A text-indexing job periodically extracts unigrams and bigrams for each category and normalizes the counts using the standard TF-IDF transformation. For content indexing, we parse source files into ASTs and collect select identifiers (e.g., names of methods, classes, variables). To compute similarity for a given change and test, we extract words from the file-system path and textual content of changed files, respectively. For the latter, we extract information directly via a simple textual diff.

The text score for a given test is the cosine similarity between words in the change and words in the test. Formally, given a test's text index D, where $D_P(t)$ and $D_C(t)$ denote the normalized word-count documents for test path and content, respectively, a set of modified files F_c for change c, and a word extraction function w, the text-similarity feature scores ϕ_2 and ϕ_3 are computed as follows:

$$\phi_2(c,t) = cosine(\bigcup_{f \in F_c} w(f.path), D_P(t))$$

$$\phi_3(c,t) = cosine(\bigcup_{f \in F_c} w(f.content), D_C(t))$$

The fourth feature is the *failure history* for the given test. Note that this feature is independent of the actual change and only depends on the test. We adopt Kim and Porter's exponential weighting function [6]. Given the observations of time-ordered test results $\{h_1^t, h_2^t, \ldots, h_k^t\}$ where h_i^t is 1 if test t passed in test execution i and 0 otherwise, the score at time k is computed as follows:

$$\phi_4^k(c,t) = \alpha h_k^t + (1-\alpha)\phi_4^{k-1}(c,t)$$

The final feature is the *age* of a test, which is also agnostic of the change. As observed and conjectured in previous work [3], new tests often fail since they exercise new and possibly churning code. However, previous features typically score new tests lower due to lack of data. We account for this cold-start problem with a binary score that boosts previously unseen tests. Formally, given the set T of observed tests, the score is computed as follows:

$$\phi_5(c,t) = \begin{cases} 1 & \text{if } t \in T \\ 0 & \text{if } t \notin T \end{cases}$$

[2]https://goo.gl/AaMEm6

4.3 Model

We integrate the features through a linear model trained using SVM$^{\text{map}}$ [17]. This algorithm is well suited for our problem formulation as it is specifically designed to maximize average precision given binary-labeled training data. To scale features uniformly, we normalize scores by their maximum values. The resulting scoring function f used to sort tests is computed as follows, where w is the trained weight vector and ϕ is the feature vector:

$$f(\delta, t) = w^T \phi(\delta, t)$$

5. EMPIRICAL STUDY

To assess the effectiveness of our approach for optimizing pre- and post-submit testing, we performed an empirical evaluation to address two research questions:

1. **RQ1**: How effectively can our approach conduct priority-based test selection, compared to using previous approaches based on an individual heuristic?
2. **RQ2**: How effectively can our approach conduct test prioritization, compared to using previous approaches based on an individual heuristic?

5.1 Data

We extracted about three months of testing results from the live automation system at Salesforce.com. We limited our analysis to the basic bucket of functional tests, containing about 45,000 tests. In total we obtained 2,000 batches of changes and their corresponding test-run results (named as batch results), including assignments of test failures to changes identified by the automated bugging system. To transform the batch results into the training data for per-change ranking, we ran Algorithm 1 as shown below.

Algorithm 1 Training data extraction algorithm

```
1: T ← ∅, passing ← batches[1].tests
2: for batch ∈ batches do
3:    for change ∈ batch.changes do
4:       failed_c = {t | (t, c) ∈ batch.failed ∧ c = change}
5:       if failed_c ≠ ∅ then
6:          T ← T ∪ {(change, passing, failed_c)}
7:          passing ← passing \ failed_c
8:       end if
9:    end for
10:   passing ← {t | t ∈ batch.test ∧ t ∉ batch.failures}
11: end for
```

In particular, batches are represented as records consisting of three fields: (1) a sequence of *changes* included in the batch, (2) a set of *tests* executed, and (3) a set of *failures* in the form of (*change*, *test*) pairs. Initially, the training data set T is empty. Next, batches and changes are traversed in order. For each change that introduced at least one failure, a training record (consisting of passing and failing tests) is added to T. Tests are assumed to be in the passing state if they passed in the last batch and have not failed for an early change in the current batch (for the first batch, all tests are assumed to be in the passing state). The underlying

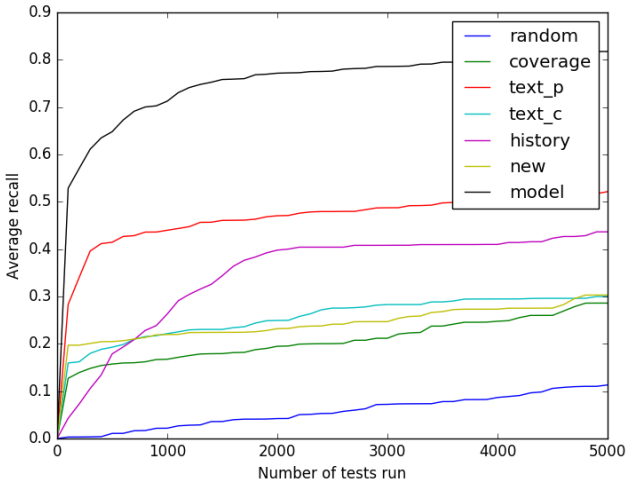

Figure 1: Average Recall

assumption is that tests do not transition result states more than once within a batch.

The preceding procedure yields 711 changes that each are associated with at least one test failure. We indexed (i.e., collected) per-test coverage and text content once, roughly at the point of 2/3 into the sequence of these 711 changes. We use changes before the indexing point as training data (440 changes) and changes after the indexing point as test data (271 changes). This procedure ensures that we apply our approach without using coverage or text data that would not be available yet.

5.2 Results and Analysis

To quantify accuracy of test selection and test prioritization, we measure recall (at various cut-off points) and APFD, respectively. The approaches under comparison are our approach (denoted as "model") and previous approaches based on coverage (denoted as "coverage"), text path (denoted as "text_p"), text content (denoted as "text_c"), history (with $\alpha = 1$), age (denoted as "new"), and random ordering (as a baseline control group, denoted as "random"), respectively.

5.2.1 RQ1

Figure 1 shows average recall across all changes in the test dataset at varying cut-off points below 5000 ($\sim$ 10%) for all the approaches under comparison. Recall is measured as the percentage of test failures detected if tests ranked before the cut-off point are executed. Each point on the curve is average recall across all changes in the test dataset at that cut-off point.

Our approach has higher average recall at all cut-off points than any other previous approach. Among the previous approaches, the one using text path has the highest average recall. There is no absolute order across the remaining previous approaches, given that their recall curves intersect, but the approach using history achieves higher average recall than the others after a slower initial increase. The remaining previous approaches seem to rank a subset of failures high, reflected by the steep rise in average recall initially, but have little predictive power for other test failures, reflected by the subsequent slow progression.

Table 1 summarizes how many tests need to be selected by each approach to achieve a given recall average. Our approach requires 72 (0.2%) top-ranked tests to detect 50% of the test failures, compared to at least 3,795 (8%) tests when using the previous approaches. Our approach requires 1,368 (3%) top-ranked tests to detect 75% of the test failures, compared to at least 19,844 (44%) tests when using the previous approaches.

Table 1: Average recall by number of tests

	15%	50%	75%
random	14% (6278)	44% (20003)	69% (30539)
coverage	0.7% (334)	34% (15431)	55% (24927)
text_p	0% (12)	8% (3795)	69% (30895)
text_c	0.1% (68)	50% (22316)	69% (30523)
history	1% (445)	16% (7419)	44% (19844)
new	0% (3)	35% (15842)	59% (26458)
model	**0% (2)**	**0.2% (72)**	**3% (1368)**

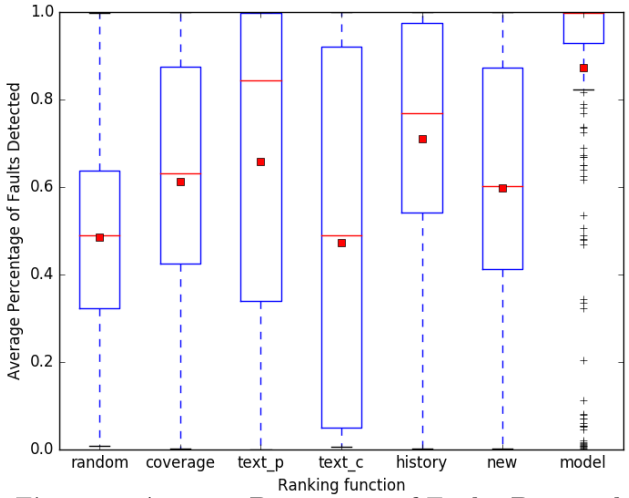

Figure 2: Average Percentage of Faults Detected

5.2.2 RQ2

Figure 2 shows boxplots of APFD scores for all the approaches under comparison. Our approach again substantially outperforms the previous approaches. The APFD median for our approach is close to 100%, with average around 85%. The previous approaches (excluding the random approach and the text-content approach) score lower, but generally outperform the random approach in terms of mean, median, p25, and p75. For the text-content approach ("text_c"), the APFD scores have midpoints around 50% and have larger variance. Among the previous approaches, the text-path approach ("text_p") achieves the highest median 84% and average 65%. The previous coverage, history, and age approaches yield similar results of midpoints slightly above 60%.

Mean AP scores induce the following ordering for the approaches: model (0.23), new (0.15), text path (0.08), coverage (0.05), text content (0.05), and history (0.02). The mean AP score for our approach, i.e., model (0.23), implies that on average about every 5th test in the ranking is a failing test until all failures are detected.

Overall, text path appears to be the strongest feature among the various individual features. The reason is that tests written to verify certain code or metadata typically use similar terms in their class or package names. As suggested by previous studies [6, 5], temporal relationships such as past failures and age of a test are also good indicators. Somewhat surprisingly, code coverage is not as strong of a

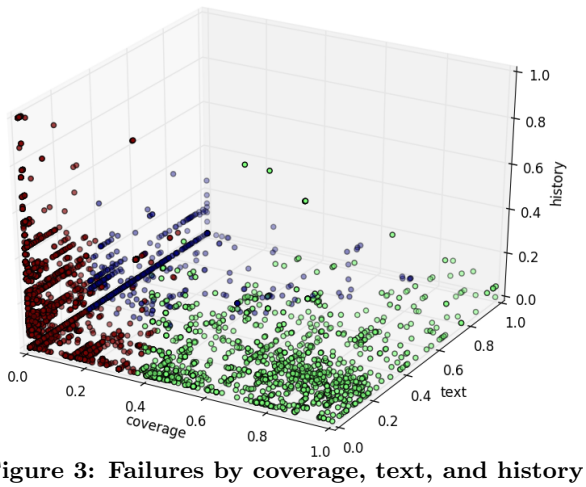

Figure 3: Failures by coverage, text, and history

classifier on its own. Such result is not due to the lack of granularity or staleness of data, but rather due to the high degree of test failures induced by non-code changes in the test suites being studied. However, the coverage feature is still important for our approach because this feature helps correctly predict test failures that are not textually or temporally related to changes.

6. DISCUSSION

Our study faces a number of threats to internal validity. The training data obtained from our automated bugging system contains some degree of noise in the form of invalid failure-to-change assignment. However, such noise affects only a small portion of the results. Another threat is potential faults in our implementation. The threat is mitigated by the use of standard tools, such as JaCoCo, and given that the heuristics in our approach are individually relatively simple. Among various threats to external validity, we evaluated our approach on only about 3 months of testing results from a single large, industrial system, which may not be representative of other testing environments.

Besides assisting test prioritization and selection, we found (previously-unexpected) side benefits of using our approach to detect flaky tests and assess the quality of our automated bug assignment. Figure 3 shows some failures from a raw data set plotted in terms of their normalized coverage, text, and history scores. Points in the upper-left corner form a small cluster of high history scores with no coverage/text-similarity to the change. Closer inspection of these failures revealed that these points correspond to flaky tests, i.e., they failed due to non-deterministic timing or environment issues. Points in the lower-left corner have very low scores for all features. These failures are typically incorrectly assigned to changes.

7. CONCLUSION

In this paper, we have presented a novel test-prioritization approach that integrates techniques from previous research via machine learning. Our initial evaluation on a large real-world dataset indicates that our approach substantially outperforms previous approaches. Our approach is efficient, practical, and designed to be well-suited for industrial settings. Our study also exposes new challenges to be addressed in future work, e.g., incorporating learning deeper into the ranker, and evaluating new features, such as better support for external artifacts packaged as archives.

8. REFERENCES

[1] R. Carlson, H. Do, and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *Proc ICSM 2011*, pages 382–391.

[2] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. CRANE: Failure prediction, change analysis and test prioritization in practice – experiences from Windows. In *Proc. ICST 2011*, pages 357–366.

[3] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proc. ISSRE 2004*, pages 113–124.

[4] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *Proc. ICSM 2001*, pages 170–179.

[5] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proc. FSE 2014*, pages 235–245.

[6] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proc. ICSE 2002*, pages 119–129.

[7] H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing, Second Edition.* Morgan & Claypool Publishers, 2014.

[8] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *Proc. ICSM 2013*, pages 540–543.

[9] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *Proc. ICST 2011*, pages 21–30.

[10] T. B. Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *Proc. ISSRE 2015*, pages 58–68.

[11] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proc. ICSM 1999*, pages 179–188.

[12] R. Saha, L. Zhang, S. Khurshid, and D. Perry. An information retrieval approach for regression test prioritization based on program changes. In *Proc. ICSE 2015*, pages 268–279.

[13] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. ISSTA 2002*, pages 97–106.

[14] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proc. ICSE 2015*, pages 99–108.

[15] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant Internet application development platform. In *Proc. SIGMOD 2009*, pages 889–896.

[16] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In *Proc. ESEC/FSE 2011*.

[17] Y. Yue, T. Finley, F. Radlinski, and T. Joachims. A support vector method for optimizing average precision. In *Proc SIGIR 2007*, pages 271–278.