2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

FastLane: Test Minimization for Rapidly Deployed Large-scale Online Services

Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila and Nachiappan Nagappan
Microsoft Research

***Abstract*—Today, we depend on numerous large-scale services for basic operations such as email. These services, built on the basis of Continuous Integration/Continuous Deployment (CI/CD) processes, are extremely dynamic: developers continuously commit code and introduce new features, functionality and fixes. Hundreds of commits may enter the code-base in a single day. Therefore one of the most time-critical, yet resource-intensive tasks towards ensuring code-quality is effectively testing such large code-bases.**

This paper presents FastLane, a system that performs *data-driven test minimization*. FastLane uses light-weight machine-learning models built upon a rich history of test and commit logs to predict test outcomes. Tests for which we predict outcomes need not be explicitly run, thereby saving us precious test-time and resources. Our evaluation on a large-scale email and collaboration platform service shows that our techniques can save 18.04%, i.e., almost a fifth of test-time while obtaining a test outcome accuracy of 99.99%.

***Index Terms*—test prioritization, commit risk, machine learning**

I. Introduction

O365 is a large enterprise collaboration service that supports several millions of users, runs across hundreds of thousands of machines, and serves millions of requests per second. Thousands of developers contribute code to it at the rate of hundreds of commits per day. Dozens of new builds are deployed every week.

Testing such a large and dynamic Continuous Integration/Continuous Deployment (CI/CD) system at scale poses several challenges. First and foremost, a massive, fully dedicated set of compute resources are required to run these tests continuously. In spite of having such dedicated resources, O365's pipelines are often unable to keep up with testing all commits thoroughly. Several enterprise testing platforms suffer from this problem [1]. This is mainly because of a growing code submission rate, and a constantly increasing test-pool size. As a result, a large fraction of commits that could potentially contain bugs go untested.

A key approach to addressing this problem is to minimize the tests to run. One can use several approaches towards this. However, with large CI/CD systems such as O365, we identify a new opportunity: Big Data storage and analytics systems [2], [3] are now ubiquitous. Modern-day services use these systems to store and process detailed logs over extended time-periods, thereby deriving rich insights about every aspect of the system development life-cycle Most relevant to our problem are large-scale commit logs and test-run logs. O365's logs, for instance, reside on big-data stores, and we routinely analyze more than one year of test and commit data in the FastLane pipeline.

We therefore ask the following question: Can we make high-confidence *data-driven* decisions towards test minimization? In other words, can we *learn* models and rules from past data that can guide more intelligent decisions on what subset of tests we need to run? There has been a large body of prior work in the test prioritization, selection and minimization areas [4]–[8]. More closely related to our work is the work by Elbaum et al. [4] which prioritizes tests based on the probability of fault existence. Our work draws inspiration from prior published work but differs in several key ways as detailed below. We analyzed O365's commit and test logs in detail, and made the following observations:

1) Some commits are more likely to fail tests than others. Not all commits are created equal. Some commits make major changes, such as changing the functionality of a network protocol. Other commits make tweaks to configuration files which may not impact core functionality at all. Some commits introduce completely new features to UI components, while others may just make cosmetic changes to the UI. This, by itself, is not a new finding: prior work has found this to hold for large open-source software repositories [9]. Nevertheless, our study helped us determine that it is indeed feasible to learn the complexity, and therefore, predict the risk associated with every commit.

2) Several tests exercise the same or similar functionality. Several developers add test-cases over time for the same code-base. In many cases, these tests may be redundant, or at the very least, related. That is, their code-coverage is largely the same. Consequently, we observe *inter-test correlations*, i.e., the outcomes of various test-pairs are heavily correlated.

3) Several tests are combinations of smaller test-cases whose outcomes are related. Such tests usually contain different code-paths that evaluate similar or same functions, leading to *intra-test correlations*. This emerges as a correlation between the test's outcome and its run-time. For instance, a specific test used the same API call multiple times with slightly different parameters. When the test fails, it tends to fail at the first API call, and therefore has a very short run-time. This emerges as a correlation between test failure and very short run-times of the test.

Based on the above observations, we designed three different approaches towards predicting test outcomes and therefore saving test resources. This paper makes the following contributions:

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00054

1) Commit Risk Prediction: Since commits vary in complexity (Observation 1), we learn *classification* models that predict the complexity of a commit and, therefore, the need to test it. Our model should predict that a commit that makes major changes is "risky", while a commit that makes a minor tweak is "safe". We can then use this label to test only risky commits while fast-tracking safe commits to deployment. We describe this in Section III-B.

2) Test Outcome-based Correlation: Since several tests have well correlated outcomes (Observation 2), we learn *association rules* that find test-pairs that pass together and fail together. For each test-pair, we then need to run only one test since we can predict the outcome of the second with high confidence. We describe this in Section III-C.

3) Runtime-based Outcome Prediction: Since some tests are amalgamations of related tests (Observation 3), we use techniques derived from *logistic regression* to estimate a runtime threshold for these tests. This threshold nicely separates passed runs from failed runs for each test. We can then use this to stop these tests after they have run for the threshold time, and predict their outcome as pass (or fail). We describe this in Section III-D.

4) Integrated Algorithm & System Design: We describe an integrated algorithm that incorporates the above-mentioned approaches into one control flow (Figure 1). We have built a system called FastLane which incorporates this algorithm (Section IV). While providing lightweight techniques, FastLane also provides threshold-tuning knobs which can be used to trade-off test-time saved with test prediction accuracy. FastLane is implemented and integrated into O365's pipelines (Section V).

5) Evaluation: We provide a quantitative evaluation of the above strategies on O365's logs collected over a period of 14 months. We find that, using such techniques, we can save 18.04% of test-time while maintaining an accuracy of test outcomes at 99.99%. Finally, we provide findings from user-studies to understand the reasons why our techniques work in practice (Section VI).

To the best of our knowledge, this is the first attempt to leverage code risk and test interrelationships using a black-box machine learning based approach for test minimization on such large-scale services. Our techniques are simple, lightweight and generic. They use well-known machine-learning and modeling techniques and so can be extended to other services as well.

II. Overview

In this section, we provide an overview of O365's development processes. We then describe our problem statement and provide an overview of our approach.

A. O365

O365 uses Git as its version-control system. It sees about 130 to 140 code commits per-day, of which around 40 to 50 commits are tested (35%). A commit varies in complexity from a single-line change in one file to major changes in hundreds, even thousands of files. The developer who creates the commit chooses one or more reviewers who review the commit. After potentially multiple iterations with the reviewers, the developer runs unit-tests on their commit. Once these tests pass, the developer runs functional tests on the commit. Our work concentrates on minimizing runs of functional tests since they are most time and resource-intensive. Henceforth, we use the term "test" and "functional test" interchangeably.

TABLE I
EXAMPLE OF THE COMPONENTS MODIFIED BY A COMMIT AND THE TEST SUITES RUN. THE THREE COMPONENTS MODIFIED ARE FUNCTIONALLY DIFFERENT. HENCE, THREE TEST-SUITES ARE RUN.

Component Modified	Test Suite Run
/src/abc	TestCreateMessageApi
/src/xyz	TestParseFormInput
/src/klm	ValidateLogEntry, WriteLogFile

Over time, test engineers and developers have built a set of rules based on code coverage and their knowledge of which code files could affect a test. These determine what set of functional tests to run on a given commit. Each rule maps a modified *component* to a set of *test-suites*, where a component is a subtree in the file hierarchy of the code-base and a test-suite is a group of related tests. Thus, depending on the components that the commit modifies, the system determines the tests-suites to run. Table II shows some example rules. Note that, by design, functional tests are independent of each other and therefore can be run in parallel.

The code-base has around 1000 test-suites. Each test-suite consists of between 1 to 1000 tests, with an average of 60-70 tests per test-suite. Each test takes between less than 1 second to more than 1000 seconds to run, with an average run-time of 6 seconds. 99.72% of all tests pass. On average, every commit runs 3000 tests. Given this set-up, some commits can take almost 30 hours to finish testing and be ready for deployment.

This extremely fast rate of development and large-scale testing causes delays to commits from being deployed worldwide. Test resources, i.e., machines on which we run tests, are limited. The number of tests to run for a commit can be large and therefore it may take a long time for all tests running on a given commit to complete. This motivates our problem and approach.

B. Problem Statement and Approach

Given the problem explained at the end of II-A, we ask the question: *Can we decrease the amount of testing required while maintaining code quality?* There are various approaches to address this question. A large amount of previous work targets the problem of test selection, i.e., how we can use code-coverage metrics and static analysis techniques to prune the set of tests to run for a given commit [4]–[8], [10]–[16]. Such techniques can help us with the problem of test selection. However, the complexity and scale of O365 require us to use additional light-weight techniques, on top of traditional test

selection techniques, to deal with the problem of minimizing test resource usage.

Fortunately, O365 uses Big Data systems to maintain logs that contain 14 months of test run-time data. These logs hold detailed information on the set of tests run on every commit, the time that each test took, and the outcome of the test. We therefore use a black-box, data-driven approach to answer the above question. We use machine-learning on test and commit logs to learn models that *predict* the outcome of certain tests so that we do not have to explicitly run them, thereby saving test resources and decreasing time-to-deployment.

To determine if such an approach is feasible, we first analyzed a fraction of the test logs to find predictable patterns. We make three observations based on this:

1) *Commit Risk Prediction ($CommRisk$):* Not all commits are equal. For instance, a commit that changes or adds a feature to code is more likely to fail tests than a commit that makes simple changes to a configuration file. We therefore use machine-learning to determine what characteristics make a commit less risky, or rather, "safe" so that we can deploy it without any testing.
2) *Test Outcome-based Correlation ($TestCorr$):* Several developers work on the same code-base and, over time, may create tests that are redundant or test very similar functionality. Consequently, several test-pairs either pass together or fail together. Such correlations exist mostly within a test-suite but can also exist across test-suites. Therefore, for a test-pair whose outcomes have been almost perfectly correlated in the past, we run only one of the two tests and predict the outcome of the second with high confidence.
3) *Runtime-based Outcome Prediction ($RunPred$):* Developers often combine testing slight variations of the same functionality into a single test. We found that for several such "composite" tests, the amount of time they took to run was very well correlated with actual outcome. These tests either fail instantly with the first variation of the functionality it tests failing, or pass after a longer duration after testing all variations. The difference in duration between passes and fails is even more pronounced when the functionality being tested is variations of the same API call, as API calls tend to inherently take longer. Some tests display the converse behavior, with tests passing early or failing after a long time. We observed that such tests typically involve network calls, and timeouts in the services they make the calls to result in their failing after a prolonged duration, as opposed to a quicker run if all the calls go smoothly and they pass. Hence, for such tests, we predict outcomes after the test has run for a certain period of time that we call a *runtime threshold*.

III. Algorithm

In this section, we first describe the log record history that we use in our analysis and predictions. We then provide details on how we learn rules for each of the three strategies outlined in Section II-B.

A. Data

Our analysis uses two data-sources:

Test logs: The O365 system stores test-logs for 14 months, and we use the logs from this entire duration for our analysis. Each log entry contains information about a $< commit, test >$ pair: it contains the start-time of the test, the end-time of the test, the host machine and the commit the test ran on, and the test outcome. The outcome can be either "Passed" or "Failed".

Commit logs: O365 stores 3 years of commit logs. Commit logs store information about each commit, namely, what time each commit was made, the developer who made the commit, the reviewers who reviewed the commit, and the files modified.

B. Commit Risk Prediction

The objective of commit risk prediction is to determine, at commit creation time, whether it is risky or safe. If it is safe, we do not test the commit, thereby saving test-time and resources.

Our approach is to use the test logs and commit logs to learn a classifier that labels a commit as *risky* or *safe*. To train the classifier, we label commits in the following way: A commit is risky if it causes *at least* one test to fail. A commit is safe if all tests run on it pass. We used a total of 133 features to characterize commits. Table II provides a brief outline. We chose our features based on inferences derived in previous work and our own findings with O365, outlined below:

- *The type of change determines how risky it is.* We capture several features related to the files that a commit modifies or adds. For instance, we capture the number of files changed, and file-types changed. We choose such features based on previous findings that certain types of file changes are more likely to cause bugs than others [9].
- *Some components are riskier than others.* We use history-based attribution to determine code hotspots. From past test-runs, for each component, we measure a "risk" associated with a component, i.e., the fraction of times that a change within a component lead to a test failing.
- *The more often a file changes, the more risky it is [1], [17].* For every file, we capture frequency of change in the last 1, 2 and 6 months, and since the creation of the file.
- *Number of contributors to a file affects risk [1], [18].* For each file, we also leverage previous work on ownership to record the developers who are major and minor owners [18] of the file, and their percentage of ownership. The features are calculated over the last 1, 2 and 6 months, and since the creation of the file.
- *Developer and reviewer history* We use features to capture developer experience and reviewer experience. A developer who has recently joined the group may be more likely to introduce a bug. Similarly, experienced reviewers will tend to find more bugs and weed them out at review-time. Some developers or their teams may work

TABLE II
A SAMPLING OF THE FEATURES USED IN $CommRisk$ AND THEIR DEFINITION

Feature	Sample Definition	#Features
File type & counts	Boolean variables per file-type capture the set of file types that a commit changes.	63
Change frequency	Maximum value, over all files changed, of the number of times a file is changed in the last 6 months.	4
Ownership	Maximum value over number of owners for each file changed.	27
Developer & Reviewer History	The number of commits made by the commit's developer so far.	37
Component Risk	The fraction of times this component has been tested and has failed at least one test.	2

on inherently "risky" parts of code, and have a greater chance of producing regressions, while some reviewers review primarily "risky" code changes. These features are calculated over the last 1, 2 and 6 months, and since the creation of the file.

Most of the characteristics we capture, such as change frequency, are at the level of a file. Therefore, to create commit-level features, we aggregate the file-specific values using operators such as `Max`, `Average`, and `Median`.

To train our model, we used a total of 133 features and a total of 16,958 commits spanning 1 year (2017). We test our model using 2 months of data, namely 3504 commits across the months of January and February 2018. We used several classifiers to evaluate our model. Table V shows the precision values for each classifier that we built. We focus on precision in the case of commit risk because it is important to not allow any high-risk change to slip into the deployment phase. The FastTree [19] algorithm, a fast implementation of gradient-boosted decision trees [20], achieved good precision, F1-score and AUC values, while remaining easily interpretable. We therefore use it in our implementation.

We found that the most important features provided were the file types. Changes to .cs files and .xml files were much more likely to cause a test to fail than changes to a .csproj or a .ini file. Our findings from the user study described in Section VI-F explain this behavior. Apart from file types, code hotspots and code ownership-based metrics also added significantly to the accuracy of our model.

C. Test Outcome-based Correlation

The second step in reducing test load is to correlate the outcome of various test-runs over time. We first describe some example reasons for the correlation and then outline our algorithm.

Our investigation revealed that there are numerous pairs of tests that always pass or fail together. We discovered the following reasons for these correlations:

The functionality covered by one test can be a finer-grained version of another broader test. One such actual pair we discovered is `TestListSize100` and `TestListSize`. `TestListSize100` tests the same functionality as `TestListSize`, except the former test runs for list sizes greater than 100. The test run logs show that all instances of these two tests either passed together or failed together. In other words, if the functionality works for sizes greater than 100, it also work with a smaller size. Thus, `TestListSize100` passing implies `TestListSize` must also pass.

Two tests depend on the same underlying functionality. One such example is `TestForwardMessage` and `TestSaveAfterSend`, both of which require the basic "send email" functionality. If that common functionality is currently failing, both tests fail together.

Two tests may be redundant versions of each other. Since multiple developers work on the same large code-base over time, they may also create tests that are redundant, or test tiny variations of the same functionality. `TestEntityUpdate` and `TestEntityRepair`: Both tests performed the same actions but written as part of two different components by two different sets of developers.

Algorithm 1 describes the $TestCorr$ algorithm. Broadly, $TestCorr$ has three steps. First, we filter the list of test-pairs that we consider to contain only pairs that have run together at least 50 times, failed together at least 20 times, and passed together at least 20 times (Line 5). Second, for every test-pair A and B, we count the number of times they both passed (or failed) on the same commit. If the ratio of this count to the count of all the times tests A and B ran together on a commit *and* test A passed (or failed), is greater than a confidence threshold C_{toc}, we learn the rule, "Test A passes (fails) $\implies$ Test B passes (fails)" (Lines 8 - 13). FastLane sets C_{toc} to the relatively high value of 0.99. Our approach towards finding correlated tests is therefore deliberately conservative.

Within the O365 dataset, in total, we found 270,550 test-pairs. This number seems surprisingly large at first-glance, but given the scale of the system, it seems quite natural to find tests that validate a subset of another test's functionality or depend on some common service.

An alternative approach would be to use rule-mining algorithms such as Apriori [21] to discover not just test-pairs but larger clusters of tests that pass and fail together. However, these algorithms are slow and, in fact, any cluster of n tests will be represented by $\binom{n}{2}$ rules of size 2. Hence, we use the lightweight, pairwise approach outlined above.

D. Runtime-based Outcome Prediction

In this section, we outline the $RunPred$ algorithm, described in Algorithm 2. First, for each test t, we use techniques derived from logistic regression to identify the duration where the *log-likelihood* of separation between failures and passes

Algorithm 1 $TestCorr$ rule generation algorithm.

Input: T: Tests run in training period, C_{toc}: confidence threshold

Output: $rules$

Initialization : $rules$ = []

1: **for** every pair of tests t_i, t_j in T where $i \neq j$ **do**
2: Let $r_{i,j}$ be #commits where both t_i and t_j were run
3: Let $p_{i,j}$ be #commits where both t_i and t_j passed
4: Let $f_{i,j}$ be #commits where t_i and t_j were both run and t_i passed
5: **if** ($r_{i,j} < 50$ or $p_{i,j} < 20$ or $f_{i,j} < 20$) **then**
6: ***continue***
7: **end if**
8: Let $pass\_fraction_{i,j} = p_{i,j}/f_{i,j}$
9: **if** ($pass\_fraction_{i,j} \geq C_{toc}$) **then**
10: Add (t_i, t_j) to $rules$
11: **end if**
12: **end for**
13: **return** $rules$

is maximized (Line 2). We call this duration the *runtime threshold* for the test t, or Δ_r^t.

Second, we check if this threshold achieves a "good separation" between failed and passed test runs for the test t. Our definition of a good separation is that the precision on the training data should be greater than or equal to a precision threshold C_{rop} (Lines 5- 12). If so, we learn a *runtime threshold* for that test (Lines 8 and 11). If the precision is lower than the threshold, we do not learn any rules for this test.

Algorithm 2 $RunPred$ Rule Generation Algorithm

Input: T: Tests run in training period, C_{rop}: precision threshold

Output: $rules\_pass$, $rules\_fail$

Initialisation : $rules\_pass$ = [], $rules\_fail$ = []

1: **for** every test t in T **do**
2: Let Δ_r^t = duration where *log-likelihood* is maximized
3: Let p_t = #runs where t's run-time $\geq C_r^t$ and t $passed$
4: Let f_t = #runs where t's run-time $\geq C_r^t$ and t $failed$
5: Let $PP_t = \frac{p_t}{p_t+f_t}$
6: Let $FP_t = \frac{f_t}{p_t+f_t}$
7: **if** ($PP_t \geq C_{rop}$) **then**
8: Add (t, Δ_r^t) to $rules\_pass$
9: **end if**
10: **if** ($FP_t \geq C_{rop}$) **then**
11: Add (t, Δ_r^t) to $rules\_fail$
12: **end if**
13: **end for**
14: **return** $rules\_pass$, $rules\_fail$

IV. System Design

In this section, we describe how FastLane integrates all three approaches into a single system design.

Figure 1 shows a flow-graph that captures the way we integrate our models. The process can broadly be divided into two stages. First, FastLane continuously learns and updates models using the three approaches described in Section III (box shown in blue). Second, as and when developers make commits, FastLane applies these models on the commits so that we can reduce the amount of testing required (boxes shown in green).

When a developer makes a commit the system uses the model that $CommRisk$ learned to determine if this commit is safe (Step 1). If not, the system determines the tests to be run on the commit using the component-to-test-case mapping described in Section II-A. Next, the system uses the rules that $TestCorr$ learn to determine test-pairs for which only one test needs to be run (Step 2). For each applicable rule, FastLane picks the test that has the lower average run-time and runs it. Based on its outcome, FastLane infers the outcome of the second test. Finally, the system applies $RunPred$, i.e., the system monitors the run-time of each test, and if this is above the pre-determined threshold, it stops the test and predicts its outcome (Step 3).

We now describe Step 4 in Figure 1. Sometimes, our models may make incorrect predictions. Moreover, code functionality and tests can change over time. Hence, we need to continuously retrain and update our models. Therefore, the system runs a fraction of all tests for which FastLane made predictions in the background. The administrator determines the value of this fraction. By running these tests in the background, and off the critical path, we make more judicial use of our test resources. By comparing the actual results of the runs with the outcomes predicted by the models, FastLane continuously validates and updates the models and their rules to be consistent with the current state.

While $CommRisk$, $TestCorr$, and $RunPred$ are generically applicable to other systems as well, we recognize that some are easier to apply than others. Not running tests on a safe commit is easy to incorporate into any test process. Incorporating $RunPred$, on the other hand, may not be easy because this requires constant monitoring of test run-time and also the ability to kill a test after the run-time threshold is reached. We therefore evaluate all three strategies independently as well and leave it to the service administrators to determine which combination of the strategies can be practically deployed in their environment.

V. Implementation

We have implemented FastLane with a combination of C# (using .NET Framework v4.5), SQL and a custom query language. We use the ML.Net library [22] to build our machine-learning models and evaluate them. Currently, the implementation is approximately 18,000 lines of code.

Since FastLane requires information about various different entities – commits, developers, reviewers and tests – a significant part of our implementation are data loaders for these different types of data. We implemented loaders for various source-control systems such as Git and others internal to our

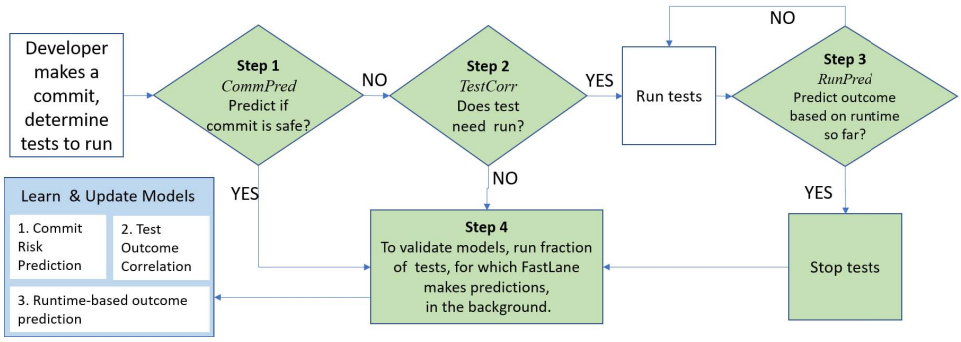

Fig. 1. The integrated algorithm and FastLane test prediction flow. This includes all three types of predictions FastLane performs (green boxes): 1. commit risk prediction, 2. test outcome-based correlation, and 3. runtime-based outcome prediction. The blue box captures the learning functionality used to continuously train FastLane.

organization. These loaders ingest source-code, code-versions and commit histories and store this data in a normalized fashion in an SQL database. Test logs, on the other hand, are stored in our organization's proprietary Big-Data storage system.

FastLane's source-data loaders and test log analyses run once a day. During the learning phase, the commit risk prediction model runs on 16,958 commits, which including file and code-review information, is approximately 36 MB of data. This takes around 10 minutes. Test logs, on the other hand, are 1.24 TB in size. FastLane takes 4 hours to learn rules for test outcome-based correlation and 1 hour to learn rules for runtime-based outcome prediction.

VI. Evaluation

In this section, we first explain the data being used in the evaluation of FastLane. Next, we describe our evaluation of how effective each of the three techniques are. Finally, we show the effectiveness of putting all three techniques together. To measure how effective FastLane is, we answer the following questions:

1) How well does each technique perform as compared to a random model?
2) How much test-time does each technique save while maintaining accuracy?
3) How much test-time does FastLane save when we put all three techniques together?

A. Data Setup

Table III summarizes our data logs. Our data broadly consists of two types of logs: commit-specific and test-specific. Our commit-specific data consists of 20,462 commits made from Jan-2017 to Feb-2018 to the O365 code-base.

We use test logs from test-runs made between Jan-2017 and Feb-2018. Our test log dataset contains around 1000 test-suites. Each test-suite contains around 60 to 70 tests on average, and the total number of individual tests is 74,213. The total number of test-runs over the entire time-period is 63,377,917. Of these, 51,102 test-runs failed and 59,619,979 passed. These fail and pass counts exclude test runs whose outcomes are neither passes nor fails, such as "not run" and "missing", which occur due to errors or resource constraints in the internal testing infrastructure. For each test-run, we record the start-time, end-time, and test outcome.

TABLE III
Summarizing our Test and Train Data

Training Data			
Log Type	**Duration**	**Size**	**#rows**
Commit	Jan-2017 to Dec-2017	35.65 MB	152,383
Test	Jan-2017 to Dec-2017	1.24 TB	54,784,935
Test Data			
Log Type	**Duration**	**Size**	**#rows**
Commit	Jan-2018 to Feb-2018	8.05 MB	36,924
Test	Jan-2018 to Feb-2018	264 GB	8,592,982

We use data between Jan-2017 and Dec-2017 to train our models, and data from Jan-2018 and Feb-2018 to evaluate them. We believe this approach is more suitable than standard cross-validation [23] because system properties change with time. As shown in Section IV, we continuously learn and update our models. Therefore the suitable approach to evaluation is to train our models using past data and test on more recent data.

B. Commit Risk Prediction

To evaluate $CommRisk$, we calculate the metrics defined in Table IV.

The Safe Precision (P_{safe}) tells us what fraction of commits that $CommRisk$ classified as safe are actually safe, while the Risky Precision (P_{risky}) tells us what fraction of commits $CommRisk$ classified as risky are actually risky.

We trained our model using various classifiers. Table V summarizes their performance. Note that the F1-score and AUC for all models are comparable. FastLane uses FastTree [19], as the interpretability of the decision trees allows us to reason about

TABLE IV
METRICS USED IN OUR EVALUATION OF $CommRisk$. TEXT IN PARENTHESES, R_{pass} AND R_{fail} APPLY TO $TestCorr$ AND $RunPred$

Metric	Definition
TP: True Positives	safe commits (passed tests) classified as safe (passed)
TN: True Negatives	risky commits (failed tests) classified as risky (failed)
FP: False Positives	risky commits (failed tests) classified as safe (passed)
FN: False Negatives	safe commits (passed tests) classified as risky (failed)
P_{safe} (P_{pass}): Safe (Pass) Precision	$\frac{TP}{TP+FP}$
P_{risky} (P_{fail}): Risky (Fail) Precision	$\frac{TN}{TN+FN}$
R_{pass}: Pass Recall	$\frac{TP}{TP+FN}$
R_{fail}: Fail Recall	$\frac{TN}{TN+FP}$
f_{ts}: Time Saved Fraction	$\frac{run-time\ of\ predicted\ tests}{total\ time\ to\ run\ all\ tests}$

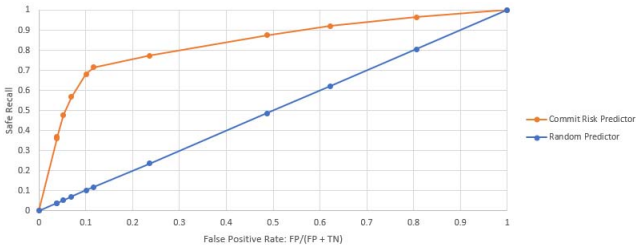

Fig. 2. AUC for the *CommRisk* model.

the model's predictions. We set the number of trees learned to 2.

We now probe a little deeper into the performance of FastTree. The Area-Under-the-ROC-Curve (AUC) seen in Figure 2 is 0.8357, which is significantly higher than the baseline that a random classifier would obtain, which is 0.5. Also, a model with an AUC greater than 0.7 for a given classification problem is considered suitable to use for that problem [27]. This shows that our classifier is very effective at predicting whether a commit will cause tests to fail (risky) or not (safe).

Ultimately though, we are interested in the amount of test run-time our model saves. This is a function of the number of commits we label as safe: the more commits our model labels safe, the more time we save since we do not run tests on these commits. However, labeling a larger number of commits as safe may also mean that we potentially miss testing risky commits. To investigate this trade-off, we now translate the output of the model to fraction of time saved, or f_{ts}. For this, we determine the time to run all tests on all commits in our test-set that we predicted are safe. We then calculate this as a fraction of total time required to run all our tests in the test-set. We then plot this as a function of safe precision.

Figure 3 shows how f_{ts} varies with safe precision. This

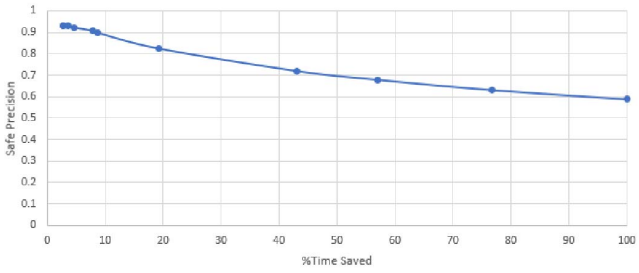

Fig. 3. Safe Precision-%Time Saved Graph for the *CommRisk* model.

curve shows that we can save 8.7% of test-time while achieving 89.7% pass precision. Note that this is at the *commit level* as opposed to the *test-case level*. The pass precision of *CommRisk* at the *test-case level* is 99.91%, and its overall accuracy on the system is 99.99% as seen in Table VIII. This overall accuracy takes into account not only the tests *CommRisk* predicts will pass and should not be run, but also the tests *CommRisk* chooses to run for commits it deems risky, thereby obtaining the correct outcomes for those tests.

C. Test Outcome-based Correlation

We now evaluate how $TestCorr$ performs. The metrics used in this evaluation are the same as those used in Table IV, except instead of *commits* being classified as *safe* or *risky*, it is now *test outcomes* that are classified as *pass* or *fail* respectively. So safe and risky precision are now referred to as pass and fail precision. We also introduce two new metrics, pass recall (R_{pass}) and fail recall (R_{fail}).

As part of our sensitivity analysis, we varied the value of C_{toc} (confidence threshold defined in Section III-C) between 1 and 0.8 to evaluate different points in the design-space. As we reduce the value of C_{toc}, we expect to find more correlated test-pairs using Algorithm 1. This implies we have more rules to apply and will therefore save more test-time. However, this may also decrease our precision values, since rules learned at lower C_{toc} values are inherently less confident than rules learned at higher C_{toc} values.

Table VI shows this effect quantitatively. We also evaluated a random straw-man for the same value of time saved, i.e., f_{TS}, and present its precision and recall numbers. As we decreased C_{toc} from 1 to 0.8, we noticed that pass precision (P_{pass}) decreases while f_{ts} increases. This is as expected. However, the fail precision (P_{fail}) initially $increases$ as C_{toc} is reduced, and reaches its peak when C_{toc} is at 0.9. This is counter-intuitive as we expect less confident rules to be less precise. However, the number of test failures predicted are of the order 10^2, while the number of test passes predicted are of the order 10^5, so fail precision is more prone to random effects than pass precision. This imbalance in number of predictions is a reflection of the low percentage of test runs which fail, as discussed in the overview of our data in Section V.

After discussions with product groups, we settled on 0.99 as the C_{toc} value for our model which provides the desired trade-

TABLE V
PERFORMANCE OF DIFFERENT CLASSIFIERS FOR COMMIT RISK PREDICTION

Model	P_{safe}	P_{risky}	**F1-Score**	**AUC**
Logistic Regression	89.68%	67.96%	0.7915	0.8357
Linear SVM [24]	80.22%	70.55%	0.7955	0.8346
Binary Neural Network	90.05%	67.67%	0.7893	0.8377
Local-Deep SVM [25]	89.54%	68.37%	0.7949	0.8238
XGBoost [26]	88.22%	68.58%	0.7950	0.8445
FastTree (Boosted Trees) [19]	**89.72%**	**68.49%**	**0.7962**	**0.8357**

TABLE VI
$TestCorr$ RESULTS SHOW HOW THE FRACTION OF TIME SAVED (f_{TS}) VARIES WITH PASS PRECISION (P_{pass}) FOR DIFFERENT VALUES OF CONFIDENCE THRESHOLD (C_{toc})

C_{toc}	f_{TS}	#rules	P_{pass}(FastLane)	P_{pass}(random)	P_{fail}(FastLane)	P_{fail}(random)	R_{pass}	R_{fail}
1	6.55%	179,046	99.97%	99.72%	84.28%	0.28%	99.98%	78.82%
0.99	**10.38%**	**270,550**	**99.95%**	**99.72%**	**88.66%**	**0.28%**	**99.99%**	**68.04%**
0.98	11.14%	356,504	99.93%	99.72%	88.85%	0.28%	99.99%	56.02%
0.95	12.26%	567,968	99.87%	99.72%	88.11%	0.28%	99.99%	41.91%
0.93	12.48%	625,684	99.86%	99.72%	88.34%	0.28%	99.99%	40.76%
0.90	12.90%	716,626	99.82%	99.72%	91.44%	0.28%	99.99%	29.12%
0.85	13.11%	779,108	99.80%	99.72%	90.12%	0.28%	99.99%	23.03%
0.80	13.20%	818,708	99.79%	99.72%	90.05%	0.28%	99.99%	21.27%

off between time saved (10.38%) and precision (99.95%). Note, this value can be tuned for any service as desired by the administrators. Overall accuracy on the system, as defined at the end of Section VI-B, and seen in Table VIII, is 99.99%.

D. Runtime-based Outcome Prediction

Finally, we evaluate the $RunPred$ technique. We use the same metrics used in Section VI-C in this evaluation. In this case, we perform our sensitivity analysis by varying the value of the precision threshold C_{rop}, as defined in Section III-D, to determine how the precision and recall values are affected as we make more aggressive predictions.

Table VII shows that we can save 1.57% of time while maintaining 99.96% pass precision, when C_{rop} is 1, which is the threshold we chose based on discussions with product groups. Again, depending on system requirements, FastLane can tune the C_{rop} value to either save more time or improve pass precision. Overall accuracy on the system when C_{rop} is 1, as defined at the end of Section VI-B, and seen in Table VIII, is 99.99%.

E. Overall Gains

Finally, we show how the models perform when we apply them to our data in different combinations. As seen in Table VIII, FastLane saves 18.04% of test-time when it combines $CommRisk$, $TestCorr$ and $RunPred$ as described in Section IV, while maintaining an overall accuracy of 99.99%.

The table also shows how much time FastLane saves when it uses other combinations of the three techniques. For instance, suppose an administrator decides to use only $CommRisk$. She will still save 8.63% of test-time.

The other point to note here is that the savings of $CommRisk$ and $TestCorr$ are fairly independent of each other. Individually, they save 8.63% and 10.38% test-time respectively. If the set of tests they predicted outcomes for were completely disjoint, they would have saved 19.01% test time. In reality, they save 17.84% together, which is only 1.17% lower.

Similarly, savings from $CommRisk$ and $RunPred$ are independent too. The sum of their savings is 10.2% while combined, they save 10.11%, very close to the sum.

On the other hand, $TestCorr$ and $RunPred$ have a larger common set of tests that they predict outcomes for. Combined, they save 10.6%, only 0.22% more than $TestCorr$ alone. This may suggest that for O365, $TestCorr$ largely subsumes $RunPred$. However, this need not be true since $TestCorr$ and $RunPred$ are very different techniques. First, in a different environment or as the system evolves with time, $RunPred$ may save on a disjoint set of tests. Second, testers in different environments may find that $RunPred$ is a more acceptable technique than $TestCorr$ since $TestCorr$ does not run the test at all and is therefore more aggressive, whereas $RunPred$ makes predictions after starting the test.

F. User Study

We performed a user-study to help confirm the validity of our models, and understand why each of our approaches works. We reached out to 100 developers, roughly 33 for each technique, and asked them whether the predictions by FastLane was correct or not — if yes, why, and if no, why not? 70 of the 100 developers responded to the study. In this section, we summarize their comments.

TABLE VII
RunPred RESULTS SHOW HOW THE FRACTION OF TIME SAVED (f_{ts}) VARIES WITH PASS PRECISION (P_{pass}) FOR DIFFERENT VALUES OF PRECISION THRESHOLD (C_{rop})

C_{rop}	f_{TS}	#tests predicted	P_{pass}(FastLane)	P_{pass}(random)	P_{fail}(FastLane)	P_{fail}(random)	R_{pass}	R_{fail}
1	**1.57%**	**162905**	**99.96%**	**99.72%**	**57.89%**	**0.28%**	**99.99%**	**23.91%**
0.99	4.34%	272451	99.92%	99.72%	61.11%	0.28%	99.99%	9.24%
0.98	4.92%	280226	99.90%	99.72%	61.11%	0.28%	99.99%	7.17%
0.95	5.35%	286552	99.86%	99.72%	58.54%	0.28%	99.99%	5.73%
0.93	5.43%	289513	99.85%	99.72%	20.51%	0.28%	99.97%	5.30%
0.90	5.64%	293381	99.85%	99.72%	43.26%	0.28%	99.97%	14.72%
0.85	5.70%	294491	99.84%	99.72%	31.75%	0.28%	99.93%	17.48%
0.80	5.77%	295867	99.83%	99.72%	34.12%	0.28%	99.91%	20.34%

TABLE VIII
OVERALL GAINS OF USING DIFFERENT COMBINATIONS OF $CommRisk$, $TestCorr$ AND $RunPred$, WHEN COMBINED AS DESCRIBED IN SECTION IV

Model	f_{TS}	%Tests Predicted	Accuracy(Predicted)	Accuracy(Overall)
$CommRisk$	8.63%	6.73%	99.88%	99.99%
$TestCorr$	10.38%	7.26%	99.94%	99.99%
$RunPred$	1.57%	2.05%	99.95%	99.99%
$CommRisk$ & $TestCorr$	17.84%	12.37%	99.90%	99.99%
$CommRisk$ & $RunPred$	10.11%	8.42%	99.89%	99.99%
$TestCorr$ & $RunPred$	10.60%	7.46%	99.94%	99.99%
CommRisk & TestCorr & RunPred	**18.04%**	**12.57%**	**99.90%**	**99.99%**

Commit Risk Prediction: Modifications to .csproj files were often simple dependency updates, and errors in these changes would be detected at build time without the need for testing. Also, changes to .ini files tended to enable an older, existing feature for different user groups. Since the feature was introduced earlier, it had already been tested, and changes to the .ini files could be checked in without testing. These reasons were mostly captured by our file-specific features described in Section III-B and Table II.

Test Outcome-based Correlation: Many tests test the same functionality in very similar ways. Some tests fail together due to outages in a service that both tests depend on, for instance, if both tests used a specific database setup. Also, tests fail together if they have similar setup processes: if the setup for one test fails, the other one will fail too. These findings are in addition to the reasons for correlation that we stated in Section III-C.

Runtime-based Outcome Prediction: The difference in durations of pass and fail runs were seen primarily in tests that make network calls. Several such test-cases validate REST APIs and start by fetching initial metadata about the REST service, or connect and make requests to a database. These requests tend to fail if the service is down, or go on to pass the test after making several time-consuming calls. The test environment being polluted by other tests is another reason for early failure.

VII. THREATS TO VALIDITY

A. External Validity: Generality of our Approach

While FastLane has shown the applicability of our techniques on O365's logs, we believe these techniques are universally applicable to any large service that uses a CI/CD pipeline, as long as we have rich logs of test and commit history. The properties we observe and build upon - commit risk prediction, inter-test correlation, and intra-test correlation - are inherent to several large-scale software development processes [8], [28]. Moreover, all our approaches use generic machine-learning and statistics techniques that do not make use of any specific properties or characteristics of O365.

B. Internal Validity

Prior work [29], [30] has investigated the problem of flaky tests and these could affect the validity of our models and rules. O365 has several flaky test detection mechanisms that use similar techniques as explored in previous work. We therefore apply these detection mechanisms prior to running our prediction model to avoid losing accuracy.

We also recognize that a trade-off exists between maintaining high precision and time-saved. To that end, we provide test administrators with various knobs for each of our techniques which can be tuned to obtain the time-saved versus precision trade-off that suits their needs best. For instance, FastLane uses $TestCorr$ with a correlation requirement of 99%. An administrator can dial this up to 100% if they want to minimize missing failures. In the case of O365, this decreases our time-saved from 10.38% to 6.55% (Table VI). On the other hand, they can turn this down too if they can accept a lower level of pass precision.

VIII. RELATED WORK

Prior work has addressed problems related to Defect Prediction, Test Selection/Minimization and Test Log Analysis.

In this section, we discuss these three aspects in the context of FastLane.

A. Defect Prediction

The features that we use in our commit risk prediction module have been derived from a vast body of previous work in the field of software defect prediction. Zimmermann et al. have shown that defects can be attributed at component or file-level [31], [32]. Our commit risk predictor therefore uses file-level and directory-level "risk" as features. Past research has also shown that process-level features are more effective at predicting software defects than code-level features [28], [33]. We therefore use such process-level features in $CommRisk$. Basili et al. investigated the ability of the classic CK Object-oriented design metrics [34] to predict fault-proneness [35]. Additionally, Nagappan et al. [17] have shown how code churn correlates with field failures. Jaafar et al. [36] investigating the effect of time found that classes having a specific lifetime model are significantly less fault-prone than other classes, and faults fixed by maintaining co-evolved classes, are significantly more frequent than others. We use similar approaches in our analysis as well, investigating changes based on when they happened and also learning from churn-based features. Bird et al. [18] have shown that code ownership (or the lack thereof) has a direct correlation with software defects. Our learner therefore also uses ownership-based features.

Our work takes inspiration from this vast body of work to develop suitable input features, and builds upon it with our specific application in mind: test minimization and resource savings. Therefore, we use commit risk prediction as a tool towards test minimization. Moreover, FastLane does not derive its definition of risk from field-level bugs and reports, but rather from test failures.

B. Test Minimization

A large body of work concentrates on the problem of test selection and minimization [7], [8], [10]–[16]. In their seminal work Rothermel et al. [5], [7] use several techniques for using test execution information to prioritize test cases for regression testing, by ordering test cases based on their total coverage of code components; ordering based on the coverage of components not previously covered; and ordering based on their estimated ability to reveal faults. The techniques studied improved the rate of fault detection of test suites. Elbaum et al. [4] advance this further and show that fine-granularity techniques outperformed coarse-granularity techniques by a small margin. Using fault-proneness techniques produced relatively small improvements over other techniques in terms of rate of fault detection and in general the effectiveness of various techniques can vary significantly across target programs. Vahabzadeh et al. [8] have recently proposed a technique that uses code instrumentation to analyze and remove fine-grained redundancy within tests.

These techniques mostly rely on code-coverage as the main criterion. While effective, applying them continuously as code and tests change can be a heavyweight process and not easily applicable to large dynamic software code-bases such as O365. FastLane, on the other hand, uses lightweight, machine-learning based techniques to perform test minimization. Moreover, FastLane, unlike previous work, gives us the ability to tune thresholds at runtime: depending on our accuracy requirements, we can turn FastLane's thresholds to achieve different time-accuracy trade-offs. Finally, previously proposed techniques based on code-coverage can generate the list of tests that are input to FastLane. Thus the two techniques can be used in tandem to obtain better test minimization. It is also important to note that FastLane differs from work on code execution classification [37], which tries to classify the outcome of a program execution as a pass or fail *after* it has completed, while FastLane predicts the outcome of a test *before* it has completed.

From an industrial stand point, in recent work at Google [1], the authors empirically leverage the relationship between Google's code, test cases, developers, programming languages, and code-change frequencies, to improve Google's CI and development processes. They find that very few tests fail and those failures are generally "closer" to the code they test and frequently modified code breaks more often. We use similar churn and test execution metrics in our analysis. Work at Microsoft [38] describes Echelon, a test prioritization system that works on binary code to scale to large binaries and to enable the testing of large systems like Windows. Echelon utilizes a binary matching system that can compute the differences at a basic block granularity between two versions of the program in binary form to prioritize tests. All such techniques can be used in tandem with FastLane.

C. Test Log Analysis

Andrews [39] first investigated the use of log files for testing, presenting a framework for automatically analyzing log files, and defining a language for specifying analyzer programs. The language permitted compositional, compact specifications of software for unit and system testing. Recent work has analyzed test logs and described the correlation between test failures and the "closeness" of code changed and frequency of code change [1]. However our concentration is on analyzing tests logs and using such insights to effectively save test-time.

IX. Conclusion

FastLane is a system that saves test-time in large-scale service development pipelines. It uses machine-learning on large volumes of logs to develop accurate models that predict test outcomes. With a combination of commit risk prediction, test outcome correlation, and runtime-based outcome prediction, we show that for a large-scale email and collaboration service, FastLane saves 18.04% of test-time while ensuring a test outcome accuracy of 99.99%.

X. Acknowledgements

We would like to thank Rob Land, Daniel Guo, B. Ashok, Sumit Asthana, Chetan Bansal and Sonu Mehta for their valuable suggestions and help building and evaluating FastLane.

REFERENCES

[1] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *ICSE SEIP Track*, 2017.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[4] S. Elbaum, M. J. Harrold, and G. Rothermel, "Test case prioritization: a family of empirical studies," in *IEEE Transactions on Software Engineering*, vol. 28, no. 10, Feb. 2002, pp. 159–182.

[5] C. C. G. Rothermel, R. H. Untch and M. J. Harrold, "Prioritizing test cases for regression testing," in *IEEE Transactions on Software Engineering*, vol. 27, no. 10, Oct. 2001, pp. 929–948.

[6] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '16, Nov. 2016.

[7] G. Rothermel, M. J. Harrold, J. V. Ronne, and C. Hong, "Empirical studies of test-suite reduction. software testing, verification and reliability," in *Software Testing, Verification and Reliability*, 2002.

[8] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *ACM/IEEE 40th International Conference on Software Engineering*, 2018.

[9] A. Hindle, D. M. German, and R. Holt, "What do alrge commits tell us? a taxonomical study of large commits," in *Proceedings of MSR*, 2008.

[10] L. Zhang, "Hybrid regression test selection," in *Proceedings of ICSE*, 2018.

[11] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proceedings. 26th International Conference on Software Engineering*, May 2004, pp. 106–115.

[12] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, Jul. 1993. [Online]. Available: http://doi.acm.org/10.1145/152388.152391

[13] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, March 2003.

[14] T. Chen and M. Lau, "A new heuristic for test suite reduction," *Information and Software Technology*, vol. 40, no. 5, pp. 347 – 354, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584998000500

[15] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stv.430

[16] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 108–123, Feb 2007.

[17] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *Proceedings of ESEM*, 2007.

[18] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *Proceedings of FSE*, 2011.

[19] Microsoft. Fasttree (gradient boosted trees). [Online]. Available: https://docs.microsoft.com/en-us/machine-learning-server/r-reference/microsoftml/rxfasttrees

[20] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, vol. 29, pp. 1189–1232, 2000.

[21] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. [Online]. Available: http://dl.acm.org/citation.cfm?id=645920.672836

[22] Microsoft. Ml.net machine learning framework. [Online]. Available: https://www.microsoft.com/net/learn/apps/machine-learning-and-ai/ml-dotnet

[23] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[24] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: primal estimated sub-gradient solver for svm," *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, Mar 2011. [Online]. Available: https://doi.org/10.1007/s10107-010-0420-4

[25] C. Jose, P. Goyal, P. Aggrwal, and M. Varma, "Local deep kernel learning for efficient non-linear svm prediction," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, pp. III–486–III–494. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042817.3042991

[26] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[27] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," 2018. [Online]. Available: http://arxiv.org/abs/1801.10269

[28] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of ICSE*, 2013.

[29] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[30] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 101–110.

[31] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *Proceedings of PROMISE*, 2007.

[32] S. Kim, T. Zimmermann, E. J. W. Jr, and Z. Zeller, "Predicting faults from cached history," in *Proceedings of ICSE*, 2007.

[33] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and statis code attributes for defect prediction," in *Proceedings of ICSE*, 2008.

[34] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," in *IEEE Transactions on Software Engineering*, vol. 20, no. 6, Jun. 1994, pp. 476–493.

[35] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," in *IEEE Transactions on Software Engineering*, vol. 22, no. 10, 1996, pp. 751–761.

[36] F. Jaafar, S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, and B. Adams, "On the relationship between program evolution and fault-proneness: An empirical study," in *Software Maintenance and Reengineering (CSMR) 2013 17th European Conference on*, 2013, pp. 15–24.

[37] D. Hao, X. Wu, and L. Zhang, "An empirical study of execution-data classification based on machine learning," in *SEKE*, 2012.

[38] A. Srivastava and J. Thiagarajan, "Predicting defects using change genealogies," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '02, 2002.

[39] J. Andrews, "Testing using log file analysis: Tools, methods, and issues," in *Proceedings of the 13th IEEE international conference on Automated software engineering*, ser. ASE'98, 1998.