Failure prediction using machine learning in IBM WebSphere Liberty continuous integration environment

Md Asif Khan
mdasif.khan@ontariotechu.ca
Ontario Tech University
Oshawa, Ontario, Canada

Akramul Azim
akramul.azim@ontariotechu.ca
Ontario Tech University
Oshawa, Ontario, Canada

Ramiro Liscano
ramiro.liscano@ontariotechu.ca
Ontario Tech University
Oshawa, Ontario, Canada

Kevin Smith
smithk6@uk.ibm.com
International Business Machines
Corporation (IBM)
United Kingdom

Yee-Kang Chang
yeekangc@ca.ibm.com
International Business Machines
Corporation (IBM)
Canada

Sylvain Garcon
SYLVAING@uk.ibm.com
International Business Machines
Corporation (IBM)
United Kingdom

Qasim Tauseef
Qasim.Tauseef@ibm.com
International Business Machines
Corporation (IBM)
United Kingdom

ABSTRACT

The growing complexity and dependencies of software have increased the importance of testing to ensure that frequent changes do not adversely affect existing functionality. Moreover, continuous integration comes with unique challenges associated with maintaining a stable build environment. Several studies have shown that the testing environment becomes more efficient with proper test case prioritization techniques. However, an application's dynamic behavior makes it challenging to derive test case prioritization techniques for achieving optimal results. With the advance of machine learning, the context of an application execution can be analyzed to select and prioritize test suites more efficiently.

Test suite prioritization techniques aim to reorder test suites' execution to deliver high quality, maintainable software at lower costs to meet specific objectives such as revealing failures earlier. The state-of-the-art techniques on test prioritization in a continuous integration environment focus on relatively small, single-language, unit-tested projects. This paper compares and analyzes Machine learning-based test suite prioritization technique on two large-scale dataset collected from a continuous integration environment Google and IBM respectively. We optimize hyperparameters and report on experiments' findings by using different machine learning algorithms for test suite prioritization. Our optimized algorithms prioritize test suites with 93% accuracy on average and require 20% fewer test suites to detect 80% of the failures than the test suites prioritized randomly.

KEYWORDS

Continuous Integration, CI, Machine Learning, Test Prioritization.

ACM Reference Format:
Md Asif Khan, Akramul Azim, Ramiro Liscano, Kevin Smith, Yee-Kang Chang, Sylvian Garcon, and Qasim Tauseef. 2021. Failure prediction using machine learning in IBM WebSphere Liberty continuous integration environment . In *Proceedings of ACM Conference (Conference'21).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Large-scale organizations have adopted the agile paradigm to maintain the quality and reliability of their products in a time-efficient and cost-effective manner. As a result, there are a growing interest in the *Continuous Integration* (CI) environments. In CI environments, developers merge code with the remote branch codebase at frequent intervals. A CI environment includes specific tasks such as version control, software configuration management, automatic build, and regression testing for new versions of the software. Regression testing involves repeating functional and non-functional tests after the submitted code change is merged with the remote branch codebase. It is executed to ensure that the codebase remains stable and the continuous development can be performed more reliably. On the other hand, regression testing needs to verify software fast and utilizing as few computational resources as possible while enhancing software quality [3].

However, in the CI environment, the frequency of code change is higher than in the systems that do not use CI. For example, Amazon engineers reported conducting 136,000 system deployments per day, averaging one every 12 seconds [14]. The frequent system builds and the repeated regression testing can increase overall test execution duration and computational requirement. For example, Google developers need to wait an extended period (45 minutes to 9 hours) to receive the testing results even though massive parallelism is available [18]. Testing services of a CI environment address this requirement in several ways. For example, the testing services limit the number of test suites to those capable of providing helpful information when a new code is merged to the remote branch codebase. Furthermore, the testing service of a CI environment can also execute tests related to specific portions of code in parallel by structuring builds so that test suites can be executed concurrently on different modules or sub-systems of the codebase. However, the test suites tend to outgrow all the available resources, although companies utilize a cluster of servers to run tests in parallel or the cloud [8].

Test selection (TS) and test suite prioritization (TP) are two well-researched techniques for improving the speed and cost-effectiveness of regression testing. TS techniques select a subset of test suites that impact a specific code submission, and TP techniques help identify issues with the code early. These techniques allow the system to optimize computational resources, provide faster feedback to developers, and improve software development and delivery speed.

Most of the state-of-the-art test datasets collected from the CI environment focus on relatively small, single-language, unit-tested projects and lack a large-scale dataset in a CI environment. For instance, the Cisco Business Conferencing [15] contains a test suite BCTS consisting of 460 test cases. Each test case contains historical execution records for the last ten runs. In total, the execution history contains results of 4600 executions only. The ABB Robotics [15] project contains 89 test case execution for ABB Paint Control dataset and 1,941 test case executions for ABB IOF/ROL dataset. We have approximately 6 million test suite execution results from two software versions combined for this research.

The most common CI dataset investigated is from Google [15], called Google Shared Dataset of Test Suite Results (GSDTSR). But the dataset is collected over 15 days only. Spieker et al. [21] pointed out that this interval is too small to evaluate. They also found that, In GSDTSR, only a few failed test cases (0.25%) occur compared to the high number of successful executions. This makes it harder to discover a feasible prioritization model as the classifiers produce more false positive prediction. This degrades the quality of the trained model.

In this paper we have worked on a novel CI dataset from IBM Liberty, an active framework for building cloud-native Java micro services. One of the version *IBM WebSphere®Application Server* has received recognition as top 50 enterprise software in the year 2020. IBM WebSphere is empowering hundreds of technologies around the world including large corporation like Amway and Canon Europe because of the scalability and robustness of the software. It is collected over 1800 commits occurring for seventeen months (March 2019 to July 2020) and contains approximately 6 million test suite execution results for two different versions.

Recent works [2, 5, 12] have shown encouraging results to motivate us to explore using ML algorithms on TP. However, the classifier used in most of these projects is not optimized. Hence in this paper, we apply ML algorithms on a large-scale dataset with a rigorous amount of training data in the CI environment to prove that ML-based TP can detect faults faster than randomly prioritized test suits.

The novelty of the work reported in this paper lies in three parts. First, this work shows that it is possible to perform TP classification using a small number of features while maintaining classification performance measures comparable to similar techniques reported in the literature. Second, our results show that the performance of ML-based classifiers improve when they are trained with large-scale dataset comparing to the classifiers trained with small dataset, and third, our research helps to determine the best optimized ML-based classifiers which are fast, easily trainable, and able to yield the best TP utilizing a large scale dataset from CI environment. In this context of this paper, we have the following contributions:

- Failure prediction using a novel large-scale CI dataset from IBM Liberty, an active java-based framework. To the authors best knowledge no analysis has been done based on this framework.
- Evaluation of the ML-based classifiers' performance trained with large-scale data set (IBM WebSphere) compared to the classifiers trained with small data set (GSDTSR).
- Prioritization of test suites utilizing optimized classifiers by training them with a small number of features and evaluating them with the classifiers trained with many features.
- Evaluation of different optimized ML-based classifiers utilizing small number of features to find the candidates which are fast, easily trainable, and able to yield the best TP when measured in terms of metrics such as accuracy, precision, recall, and F_1 score by utilizing a large scale dataset from CI environment

The remaining of the paper is organized as follows. The problem statement is defined in Section 2. We introduce the related state-of-the-art TP techniques in Section 3. Section 4 discusses the data collection, data processing workflow, and the use of different machine learning algorithms. In Section 5, we present our experimental setup and discuss the results. Finally, we discuss the lessons learned in Section 6, and Section 7 concludes the paper.

2 PROBLEM STATEMENT

The goal of this paper is to train and optimize ML-based classifiers using large-scale dataset which will outperform the randomly prioritized test suites in TP. This approach can be formalized by three related research questions as follows:

- **RQ1**: Is it possible to perform TP with a small number of features while maintaining classification performance measures comparable to similar techniques reported in the literature that use a large number of features?
- **RQ2**: Does the performance of ML-based classifiers improve when they are trained with large-scale dataset comparing to the classifiers trained with small dataset?
- **RQ3**: Which are the best optimized ML-based classifiers which are fast, easily trainable, and able to yield the best TP when measured in terms of accuracy, precision, recall, and F_1 score by utilizing a large scale dataset from CI environment?

In version control system, developers sends their latest code changes as a operation called *commits* to the remote codebase. Each commit contains a set of important information related to the code changes such as author name, line changed, timestamp etc. Therefore, given a set of T test suites executed for every commit C of a large set of commits within a certain duration D, the goal of this paper is to find prioritized T using an efficient ML-based test case prioritization with regards to evaluation metrics.

3 RELATED WORK

Machine learning algorithms are receiving increased attention from the perspective of software testing. Busjaegaer and Xie [5] use supervised machine learning to prioritize test suites in an industrial setting. They extract various testing results and training on the Support Vector Machine model, and show that their supervised TP technique outperforms all comparison techniques. This work makes a strong foundation to define a general framework to learn to prioritize test suites automatically. This study also compares different techniques, such as random prioritization, test history, and coverage prioritization. However, Busjaeger and Xie have used TS after prioritizing test suites and report different recall values.

Spieker et al. [21] perform priority-based testing based on the approach of Busjaegaer and Xie. They apply reinforcement learning and artificial neural network to implement a lightweight learning method. Their technique utilizes both TS and TP in the suites. The main idea is to prioritize test suites first and then select the topmost tests under a certain time threshold.

Yoo et al. [24] utilize Google testing datasets to describe a search-based approach to apply TS and TP techniques where they use dependency coverage, among other features. They described that at Google, the developers working in a CI environment first test their code in some initial local build before submitting to the remote branch codebase. This step is called *pre-submit testing*. The pre-submit testing allows the developers to detect as many integration errors as possible before entering the remote branch, breaking builds, and delaying the desired fast feedback [24]. Sometimes the developers who adopted CI environment to scale and speed up their development heavily rely on pre-submit testing. The over-reliance on testing can make the system overloaded by consuming excessive resources. Simultaneously, rigorous testing by the test services after a code merge (*post-submit testing*) also requires substantial computational resources. Developers require to perform testing faster to release the application code faster, and this helps the integration of various code commits quicker. This approach, however, has evaluated cost-effectiveness in a pre-submit testing environment.

Bagherzadeh et al. [2] pave the way to use reinforcement learning (RL) to prioritize test suites in a CI context. They implement RL technique using three alternative ranking model to test the interaction between the CI environment and a TP agent. Their RL solutions provide better accuracy over state-of-the-art RL-based solutions.

Lima et al. [15] and Malhotra [16] perform comprehensive systematic review, which has helped us to investigate how ML-based TP techniques are used for fault prediction, fault proneness of modules/classes. Malhotra considers 64 studies from a period between 1991 and 2013 and identify the following categories of ML techniques: Decision Trees (DT), Bayesian Learners (BL), Ensemble Learners (EL), Neural Networks (NN), Support Vector Machines (SVM), Rule-Based Learning (RBL), Evolutionary Algorithms (EA) and miscellaneous. The top techniques among these categories are C4.5 (a DT), Naive bayes (a BL), Multilayer Perceptron (a NN), SVM, Random Forest (an EL). Furthermore, the author has also categorized the metrics used by these studies into procedural metrics (traditional static code metrics suites, and size metrics as lines of code), object-oriented metrics (cohesion, coupling, and inheritance), hybrid metrics (a combination of procedural and object-oriented metrics), and others. The author concludes that machine learning techniques achieve adequate prediction capabilities. The Random Forest model outperforms all the other ML models, and the models that utilized Naïve Bayes and Bayesian Networks outperform the remaining techniques. Thus, We have selected Random Forest, Naïve Bayes, and Logistic regression for our own ML-based TP technique.

Some work exist [11, 20] on TP in CI environments. These works discuss the use of continuous testing and TP in continuous integration to help organizations to reveal failures faster. Test case prioritization is performed using different modules of the codebase, i. e. test suites or test suites (collection of test suites). Developers often prioritize test suites in the CI environment due to the high frequency of code merge and a large collection of test suites. Prioritization of a large number of individual test suites is impractical in most suites for CI environment with large dataset. However, some prior researchers [8, 24] also consider prioritization of commit. Two types of commits exist: 1) Intra-commit (prioritizing tests inside commits), and 2) Inter-commit(prioritizing tests among multiple commits). However, test suites are often small in size and can be quickly executed in inter-commit TP. Therefore, reordering these test suites typically cannot produce large reductions in feedback time. Furthermore, these commits often have dependencies among themselves that make reordering error-prone. When we consider inter-commit, multiple test suites related to individual commits are queued up until a clean build of the system is available for their execution over different computing resources. This queuing process sometimes leads to an impractical waiting period [10] for the test results. We have considered these issues and test suites, also known as buckets in our TP technique.

Several industrial case studies have focused on TP. For example, Microsoft used TP for Windows [7, 22] and Dynamics Ax [6]. Google evaluated TP to optimize pre- and post-submit testing for a robust, frequently changing code repository [8]. Cisco explored TP to reduce long-running video conferencing tests [17].

4 USING ML ALGORITHMS FOR TEST SUITE PRIORITIZATION IN LARGE-SCALE CI

In the CI environment, a developer usually pushes code changes as commits to the remote branch codebase to merge after adding a new feature or a fix. At the same time, other developers also send their changes as a stream of commits in the CI environment. If there is no prioritization, then the regression test suites related to the changed code is executed in random order. The test suites are all scheduled in the CI pipeline to get executed once the resources are available. The system can evaluate the performance (e.g., how fast the faults are detected or how much coverage the test suites have gained) using different metrics after the test suites are executed. ML-based TP techniques can be introduced to facilitate the prioritization of the test suites. We apply TP techniques to a large-scale test executions dataset of a CI environment to gain insight into which ML-based classifiers are the most helpful for us in the CI environment using small number of features.

We first find the oldest commit and then move towards the newest commit to collect the test suite execution information. As

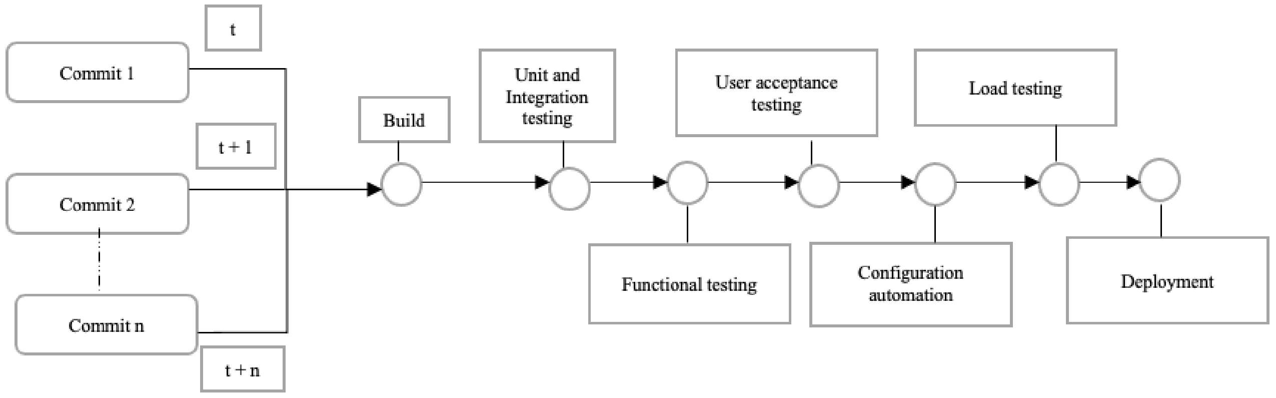

Figure 1: Commit specific test execution in a CI environment

part of the workflow, we also collect the total number of tests that are within the test suites, test verdicts and test duration for every test suite triggered for a commit and then apply TP. Therefore, we separate our test case prioritization workflow into few phases: 1) Data collection, 2) Data summarization and 3) Filtering. Afterward, we apply TP using ML algorithms. A simplified architecture of a CI environment where we apply our ML-based TP techniques is shown in Figure 1.

4.1 Data Collection and Feature Selection

4.1.1 The IBM WebSphere Liberty Data Set. We have worked with the novel dataset that comes from the IBM WebSphere Liberty, which is a fast and composable Java-based application server to create applications and cloud-based services [1]. WebSphere Liberty uses Java EE and MicroProfile core of the Open Liberty project. There are two different builds of the Liberty software, *Open Continuous* and *Continuous* build. These two builds rely on different incoming continuous integration code streams. There are also four different test file types for each build. They are: Functional Acceptance Test (FAT), Build Verification Testing (BVT), Unit test (Unit), and Load test. FAT is the core part of IBM testing, and differs from Unit tests because FAT tests execute on a running OpenLiberty image. We will use the continuous build FAT test file for our testing purpose.

The continuous build dataset contains around 4 million entries while open liberty contains around 2 million entries. They have the following features:

- *Parent job name* is the name of the main job, and all results with the same parent job are considered to be part of the same build.
- *Job name* is the name of the child job for a parent job. The child jobs are how the test executions are parallelized. Each parent job is distributed among multiple children. Each child runs a test suite that contains various test suites. Child name includes their related parent job name.
- *Bucket name* is the name of the test suite.
- *Bucket type name* is the name of the test type.
- *Duration* is the length of time to execute the test suite (bucket) in milliseconds.
- *Job UUID* is a unique id attached to the child builds.
- *Test total* is the total number of tests that ran within a test suite.
- *Test failures* is when a test fails its designated verification (an assert failed). When the number of test failures is greater than one, then the test suite is considered to have *Failed.*
- *Test errors* is when the test throws an exception at an unexpected time and thus is faulty. When the number of test errors is greater than one, then the test suite is considered to have *Failed.*
- *Failed* based on the values of *test failures* and *test errors*; this shows either true or false status.

Table 1 shows total values of some important features of both *Open Continuous* and *Continuous* build.

Table 1: Data Set description

Features	WebSphere Liberty	Open Liberty
test suites	3997133	1901934
Test Run	229977494	97891276
Failures(F)	76230	34240
Errors (E)	93081	26250

One of the biggest challenges when dealing with TP for CI is the multidimensional nature of the data. We have to represent test suites as a vector of features. Developers gather huge amount of data with various features from the CI environment. However, not all the features are useful to train the ML models. The selection of feature varies based on the goal of the machine learning model. For example, if the goal of TP is to detect faults then test suite suite size will be a good feature. This means larger test suites are more fault prone. Hence, in this work, We have used two features, namely *Test total* and *Duration* to train our ML models.

4.1.2 The Google Data Set. The Google Shared dataset of Test Suite Results (GSDTSR) contains of over 3.5 million test suite execution results. This data set is collected over a period of 30 days from a sample Google product. The first 15 days of data are being used as there are discontinuation in the later days. Spieker et al. converted the original GSDTSR dataset into common file format and we have used this version for our study.

The converted GSDTSR dataset contains 1.2 million data with the following features:

- *Cycle* The CI cycle number this test suite execution belongs to
- *Id* Unique numeric identifier of the test execution
- *LastRun* Prior last execution of the test suite as date-time-string
- *Duration* is the length of time to execute the test suite (bucket) in milliseconds.
- *Verdict* Test execution result. It the test suite fails then the verdict is *True* otherwise the verdict is *False*
- *LastResults* List of previous test execution results in ascending order. Lists are delimited by [ ].

Table 2 gives an comparison of the two data sets' structure.

Table 2: Data Set description

Data set	Test Suites	CI cycles	Verdicts	Failed
Google GSDTSR	5,555	336	1,260,617	0.25%
IBM Open Liberty	1,051	1847	1,901,934	1.02%

4.2 Data summarization and filtering

The original dataset contains individual test suite execution information. However, we want to know the information in terms of the parent job. For example, we want to know the number of test suite executions, duration, failures, and errors of the parent job "20200733-1501". So, we count the number of test suite executions for each parent job to obtain total test suite execution. Then, We sum up the duration, test total, test failures, and test error values for all the test suite executions under the same parent job.

We have used the summarization technique on 150,000 entries and produced a summary of the parent job. For example, The parent job "20200733-1501" runs for around 40 hours and contains 545 test suite executions. Among these executions, one fails, and there is no error. So the total fail and error (FE) are 1. We then calculate the FE ratio, which is 0.18 in this instance.

We aggregate the average number of buckets execute by each parent job and find that at least 49 parent job contains over 700 buckets. In summary, the average buckets per job is 697, duration is 51.7 hours, failure & error is 7.08, and failure ratio 0.98%

From our summarization result we also find that certain jobs contains higher percentage of test suites than the average. Based on our observation, we have filtered out the jobs which contains 50% more test suites than the average (i.e. 1050).

We filter out all the test suites that are not FAT type. We also develop a custom data visualization script to sanity check and identifying bad data. It reveals that there are few examples where test duration is negative. The potential reason could be time zone difference as parallel tests are run on different locations. Sometimes due to technical reason or code error, the test suites executions are halted, resulting in a duplicate run of the same test suites.

The number of test suites reduced by 7% after applying the filtering of the non-FAT test. The number of test suites further reduced to around 2% after removing duplicates. Therefore, the filtering has reduced the number of test suites to approximately 9%.

4.3 Using ML algorithms for TP

We apply five different ML algorithms for TP and compare the performance in terms of metrics such as accuracy, precision, recall, F1 score, and percentage of faults detected, like in previous studies [5, 12, 19]. One technique prioritizes test suites randomly, and the other five use machine learning. The machine learning algorithms are Random Forest (RF), Naive bayes (NB), Decision Trees (DT), k-Nearest Neighbors (k-NN), and Logistic regression (LR). All these algorithms are commonly used in the related literature [16, 23]. Table 3 describes and summarizes each ML-based TP technique.

Table 3: MACHINE LEARNING ALGORITHMS

Technique	Description
Random	Prioritization of test suites randomly
Random Forest	Prioritization of test suites with Random Forest classifier.
Naive bayes	Prioritization of test suites with Gaussian Naive bayes classifier
Logistic regression	Prioritization of test suites with logistic regression classifier
Decision Trees	Prioritization of test suites with decision tree classifier
k-Nearest Neighbors	Prioritization of test suites with k-nearest neighbors classifier

In a CI environment, sometimes multiple parent jobs execute the same commit. One test suite might pass under one parent job but might fail under another parent job even though the code base is the same, however, they are running under different environments. The

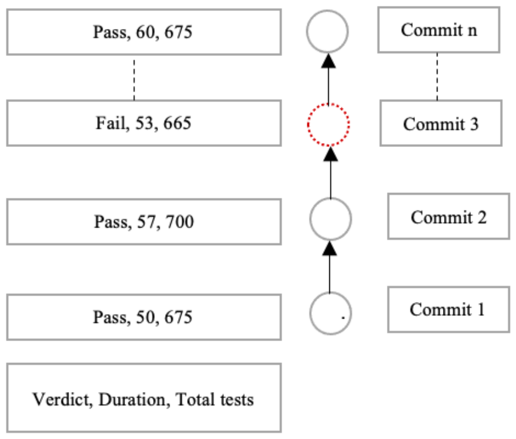

Figure 2: Data collection steps of commit specific test execution

goal is to take into account not only the commit level information but also the execution of them under various parent job.

First, we check the oldest commit. Then, we check if the commit is executed under multiple parent job. If it is true, then we run all the test suites belong to those parent job and collect duration in hours, test suites within each test suite, and result of each parent job. Then we move on to the next commit. Figure 2 illustrates the version control history with commit specific test execution information. The circles represent commits and their color represents the test verdict (red represents a failed parent job). We train the machine learning classifiers using 80% of the dataset and test with the rest of the 20% of the dataset. We record the accuracy, precision, recall, and F_1 score values based on the output. Then, we move on to the next commit and follow the same procedure until we reach the newest commit. Every time we are moving to the new commit, the number of builds is increasing as we are considering historical commit information. We are also assuming that the number of test suites remain unchanged and faults occur at random in the test suites.

To evaluate the percentage of faults detected of ML-based TP techniques, we need to prioritize test suites based on some criteria. In our ML-based TP technique, we have tried to find the test suites which have higher probability of failure. We have used *class probabilities* from scikit-learn package to calculate the probability of failure. The class probabilities estimate the probability for a test to fail and pass as a tuple of $\{p_1, p_2\}$, where p_1 is the estimated probability of t to fail and p_2 is the estimated probability for t to pass. Furthermore, We sort the test suites executions in descending order based on p_1. In ML, such an approach is called pointwise ranking [13]. In scikit-learn the class probabilities can be calculated with function predict_proba().

5 RESULTS AND ANALYSIS

5.1 Experiment Set-up

5.1.1 Hyperparameters for the ML models. Some machine learning models require certain hyperparameters before implementation and are usually set by the user. Finding a good set of hyperparameters is a difficult task, however, there are few techniques that can make the process easier. Grid search is a well-known technique to find optimum hyperparameters. Grid search is supplied with a set of candidate values, and it uses different combinations of those candidates to train the ML model. After the training, the combination of candidates that outputs a high score (i.e., accuracy, precision, or recall) is selected as optimum hyperparameters. However, the efficiency of grid search decreases with respect to the nature and the number of parameters. Grid search also becomes slow if there are many hyperparameters and the search space is huge. Random search, in many suites, is more practical because it can be implemented in parallel computers and adding new trials or removing broken ones on the runtime is possible. These things are not feasible in the grid search.

Bergstra & Bengio [4] showed that random search performs better than the grid search for hyperparameter optimization. They have compared these two hyperparameter optimizations on several data sets using several machine learning techniques. They have found that with the same computational resources as the grid search, random search finds better models in less time. They have also shown that random search performs better than grid search in high-dimensional spaces.

One important observation is that initially, the random search does not require the candidate sets, which are actually selected at runtime, unlike the grid search. With the required components, *RandomizedSearchCV* randomly searches over the given parameters by implementing a *fit* and a *score* method. These methods are optimized by cross-validating searches over different parameter settings. Finally, after the specified trial, the candidates which produce the best output in the objective functions are selected.

We have used *RandomizedSearchCV* in *scikit-learn* on our sample data and labels to avoid data leakage. We have to provide the following components to implement the random search.

(1) Precision as the objective function to calculate the quality of the candidate set;
(2) For the search space we have used 10, 000 of our sample data;
(3) Total number of trials varied between 100 - 1000 based on different ML models.

We have obtained the following hyperparameters after utilizing *RandomizedSearchCV*.

- Logistic regression
  - Algorithm that is used in the optimization problem (solver): liblinear,
  - Tolerance for stopping criteria (tol): $1e^{-2}$
  - Maximum number of iterations permitted for the solvers to converge (max_iter): 1000,
  - Shuffle the data (random_state): 0
- Naive bayes

  - Part of the largest variance of all features that is added to variances for the stability of the calculation (var_smoothing): 0.81,
- Random Forest
  - The number of trees present in the forest (n_estimator): 100,
  - The function to calculate the quality of a split (criterion) : gini
  - The minimum number of samples necessary to split an internal node (min_samples_split): 2
  - The randomness of the bootstrapping of the samples used when creating trees (random_state): 0

5.2 Results

We want to find out how the performance of various ML algorithm changes with the increase in number of test suites. We have used two large scale CI data sets discussed earlier, train our ML algorithm and record evaluation metrics. We assume that each parent job is running one commit and collect only test suite level information for each job. Values of four evaluation metrics, namely Accuracy, Precision, Recall, and F_1 score are listed in Table 4.

In our implementation, we considered *Accuracy* as the total number of correctly classified test suite result divided by the total number of test suite executions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Here,

- True Positive (*TP*): Test suites that are classified as "failed" and the actual outcomes are also "failed".
- True Negative (*TN*): Test suites that are labeled as "passed", and their actual outcomes are also "passed".
- False Positive (*FP*): Test suites that are labeled as "failed", however, their actual outcomes are "passed"
- False Negative (*FN*): False negatives test suites are labeled as "passed", however, their actual outcomes are "failed".

Precision is the ratio of correctly identified failed test suites divided by the total number of test suites labeled as belonging to the failed class. High precision means that a classification model returns substantially more relevant test suites results than irrelevant results.

$$Precision = \frac{TP}{TP + FP}$$

Recall is the number of correctly identified failed test suites divided by the total number of test suites actually belong to the failed class. High recall means that a classification model returns most of the failed test executions results.

$$Recall = \frac{TP}{TP + FN}$$

We consider, F_1 *score* as the weighted average of of the precision and recall. It can also be interpreted as the harmonic mean of precision and recall. The best value for the F_1 score is 1 and the worst value is 0. For any ML model, the closer the value of F_1 score towards 1, the better the model is.

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Table 4: Evaluation metrics of ML models

Metrics	LR	NB	RF	DT	k-NN
Accuracy					
GSDTSR	0.92	0.91	0.93	0.93	0.93
WebSphere Liberty	0.93	0.92	0.92	0.94	0.92
Precision					
GSDTSR	0.90	0.90	0.92	0.93	0.93
WebSphere Liberty	0.91	0.90	0.92	0.92	0.91
Recall					
GSDTSR	0.92	0.91	0.93	0.93	0.93
WebSphere Liberty	0.92	0.92	0.92	0.93	0.91
F_1 **Score**					
GSDTSR	0.89	0.88	0.92	0.93	0.93
WebSphere Liberty	0.90	0.91	0.90	0.91	0.91

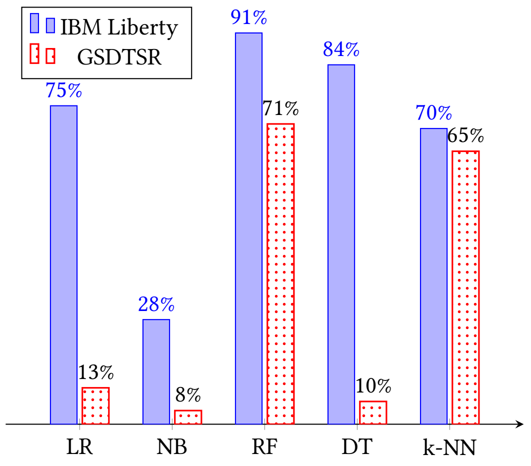

Figure 3: Precision metric to predict Failed tests

For random forest classifier, Malhotra [16] showed minimum accuracy value of 55% and maximum value of 93.5% with mean value of 75.63% and median value of 75.94% respectively. Our result from Table 4 shows a better result for both GSDTSR and Open Liberty dataset, with minimum accuracy value of 91% and maximum value of 93%.

We can also see from Table 4 that, the precision score ranges between 90% and 93% for GSDTSR while WebSphere Liberty ranges between 90%-91%. The minimum recall value for both GSDTSR and WebSphere Liberty is 91% while the maximum value is 93%. In

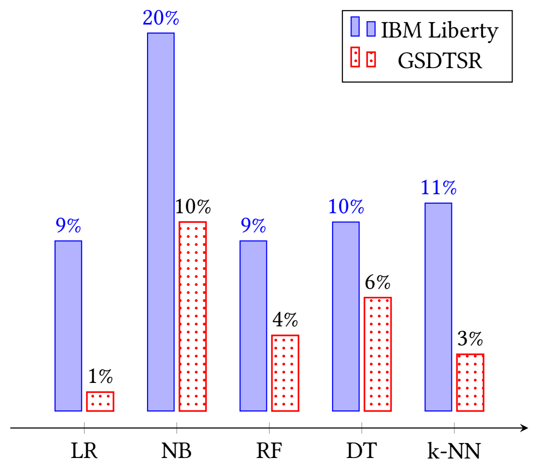

Figure 4: Recall metric to predict Failed tests

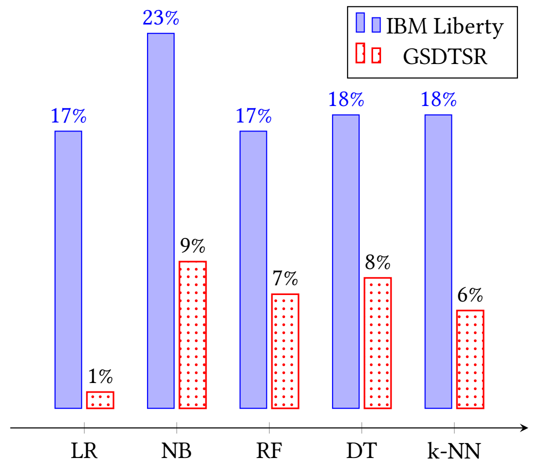

Figure 5: F_1 Score metric to predict Failed tests

terms of F_1 score, WebSphere performs better for LR and NB. On the other hand for tree based classifier (RF, DT) and k-NN, GSDTSR performs better than WebsSphere Liberty.

To analyze further the values of precision, and recall we show the confusion matrix of the whole dataset. *Confusion matrix* is a binary classifier proposed by Fawcett [9] as shown in Figure 6. Confusion matrix can be utilized to see the distribution of TP, TN, FP, FN values when calculating different evaluation metrics.

It is clear from Table 4 that all machine learning models produce similar results in terms of accuracy, precision and recall. The notable difference we can observe when we consider F_1 score. The tree-based models perform, on average better than the other models. One of the reasons is the nature of the dataset and the problem we are dealing with. As we are solving the binary type classification problem (Failed or Passed) using an unbalanced dataset in which tree-based classification models perform better. We can say that, on average, Random forest and Decision tree classifiers outperform other models. This does not guarantee that these classifiers will

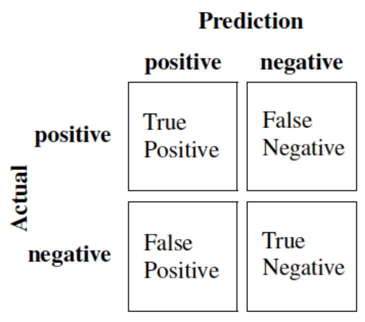

Figure 6: A two-by-two confusion matrix ([9])

perform similarly in similar projects, however, there is a higher probability of producing a better result. We can also see that NB has the lowest false positives comparing to other ML models, which results in low F_1 score.

Table 5: Confusion matrix for ML models

ML Model	TP	TN	FP	FN
LR	724929	16076	56582	5566
NB	710830	24668	48071	19584
RF	726470	15277	57600	3806
DT	725437	16991	56220	4505
k-NN	716528	19049	53655	13921

Rothermel et al. introduced a metric called, *Average Percentage Of Faults Detected* (APFD) to measure a test suite's rate of fault detection. APFD values range from 0 to 1. A higher APFD value indicates better failure detection rates.

Let T be a test suite consisting n test suites and let F be a set of m faults identified by T. Let TF_i be the first test suite in ordering T_0 of T which identifies fault i. The APFD for test suite T_0 can be calculated by the following equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + ...TF_n}{nm} + \frac{1}{2n}$$

We illustrate this metric using an example. Consider a parent job with 10 test suites, T_1 through T_7, such that the program contains seven faults, F_1 through F_7 detected by those test suites, as shown by the table 6. All the test suites detect faults except for test suite T_5.

Using the equation above the APFD for random test suite execution:

$$APFD = 1 - \frac{4+3+2+2+1+2+7}{49} + \frac{1}{14} = 0.50$$

Then, the APFD for Random Forest classifier:

$$APFD = 1 - \frac{6+1+3+3+7+3+5}{49} + \frac{1}{14} = 0.64$$

Similarly, we calculate he APFD value for Naive Bayes is 0.66 and 0.50 for Logistic regression. Higher APFD value of RF means, it is predicting more faults than the random execution after each test execution. For example, if the tests are executed sequentially, when T_1 is executed it reveals $\frac{18}{193} = 9.33\%$ of of total faults. Then, T_2 reveals 11.4% of faults. Hence, the test suite reveals 20.73% of total faults after executing T_1 and T_2. The goal is to prioritize tests that reveal more faults as early as possible.

Figure 7 shows that the all three machine learning classifier has higher area under the curve than the random test executions. The Gaussian NB classifier detects 75% of the faults after five tests, while random execution reveals 50% of the faults at the same time. On the other hand, Figure 7 shows that the RF classifier detects 80% of the faults after five tests, while random execution reveals 50% of the faults at the same time. LR classifier detects faults linear to the random execution of test suites for the initial two tests. However, the fault detection increases to 78% after five tests while the random execution of test suites reveals 50% of the faults at the same time, as shown in Figure 7 .

If we look at the Table 6, we will find that T_6 and T_7 have the highest number of faults. The ML models that execute these two test suites can detect fault early. We have found that RF machine learning technique prioritizes test suites to detect faults early compared with other ML techniques.

Table 6: Sample tests to calculate APFD

Test suite/ Fault	T_1	T_2	T_3	T_4	T_5	T_6	T_7
F_1				X			X
F_2			X			X	X
F_3		X				X	X
F_4		X				X	X
F_5	X					X	X
F_6		X				X	X
F_7	X		X			X	X

6 DISCUSSION

In this paper, we compare the performance of five TP techniques based on the machine learning algorithm. We measure five different evaluation metrics for each technique: accuracy, precision, recall, F_1 score, and the average of the percentage of faults detected.

6.1 Challenges

We have faced several challenges when dealing with the large-scale dataset. Defining outliers is a significant challenge when working with a lot of data. For example, we find that the average number of

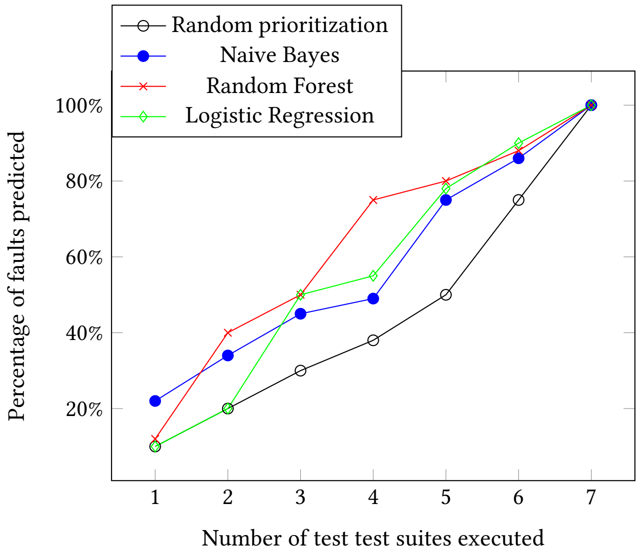

Figure 7: Percentage of faults predicted by TCP using ML

test suites per parent job is around 700, however, certain parent jobs contain 1400 test suites. After investigation, we find that sometimes due to technical error, certain jobs are halted and then restart again, causing them to rerun previous test suites and record a large number of test suite execution. Data summarization and filtering is another challenging aspect. In a large-scale dataset, it is normal to have incomplete and inconsistent data. We sit down with developers to clarify the potential anomalies. The multidimensional nature of big data also poses a challenge as not every feature will be useful for test suite prioritization.

6.2 Research questions

- *For research question RQ1*: Our results pointed out that it is possible to perform TP with a small number of features while maintaining classification performance measures comparable to similar techniques reported in the literature that use a large number of features. Our optimized ML-based classifiers achieved average Accuracy and F1 score above 91% similar to the classifiers reported in the literature.
- *For research question RQ2*: Our results show that the performance of ML-based classifiers improve when they are trained with large-scale dataset comparing to the classifiers trained with small dataset. We find that the performance of ML-based TP techniques is increased when more test suites are executed. Additionally, the performance increases when more data are being added to the training dataset. This is inline with our expectations because we know that data is one of the key components in making ML algorithms efficient to use. The values of evaluation metrics in our results validate this key aspect.
- *For research question RQ3*: Among the five ML-based classifiers, the random forest and decision tree performs TP better than the other classifiers when measured in terms of accuracy, precision, recall, and F_1 score utilizing two large scale

data sets from CI environment. The reason is that the RF model is suitable for the large-scale unbalanced dataset because it minimizes the overall error rate. When there is an unbalanced dataset, the RF model assigns a low error rate to the larger category while the smaller category is assigned a larger error rate. Hence, the RF model performs better with our unbalanced dataset.

6.3 Threats to Validity

There are a few limitation in our experiment. Firstly, we tried to include as many test suite result as possible. However, test suites were removed from analysis based on our filtering process. This arguably distorts the test prioritization results and empirical studies can be done to confer our findings.

Another threat to internal validity is the influence of random decisions on the results. To mitigate the threat, we have provided average and median results of your evaluation metrics..

We do not know the actual faults in the Liberty software and we assumed a failing test represents a unique fault. This means that the average percentage of faults detected values are just estimates. Related studies [5] have made the same assumption that a failing test reveals one fault.

We applied test suite prioritization to a unbalanced dataset from a single project only. This points out that external validity can be affected. We have plan to compare our ML models with dataset containing different ratio of failures.

Finally, machine learning algorithms are sensitive to their hyperparameters and a good hyperparameter set for one dataset and environment might not work for as well for different dataset. During out experiment, we have optimized our hyperparameter as real-world practice. So, our hyperparameters might not perform the same in different dataset. Moreover, all the machine learning algorithms that we have used are supervised techniques.

7 CONCLUSION

We have evaluated two large-scale industrial CI dataset from Google and IBM with five machine learning algorithms and analyzed their results. In our experience, all of our optimized ML-based classifiers have achieved average Accuracy and F1 score above 91%. In terms of F_1 score, the larger data set IBM WebSphere with more CI cycle outperforms the Google Data set. All of the optimized ML-based classifiers require 20% fewer test suites to detect 80% of the failures than the test suites prioritized randomly.

Our ML-based TP techniques require additional verification even though our results show positive outcomes. We use only two features to train our ML models. Adding more features will have an impact on the performance of the ML models. We have worked exclusively with the continuous build dataset for our testing purpose. However, we have plans to use the open continuous dataset to investigate ML-based TP techniques' performance.

REFERENCES

[1] [n.d.]. WebSphere Liberty - Overview. https://www.ibm.com/ca-en/cloud/websphere-liberty

[2] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2021. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering* (2021).

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 356–367.

[4] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research* 13, 1 (2012), 281–305.

[5] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: an industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 975–980.

[6] Ryan Carlson, Hyunsook Do, and Anne Denton. 2011. A clustering approach to improving test case prioritization: An industrial case study.. In *ICSM*, Vol. 11. 382–391.

[7] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. 2011. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 357–366.

[8] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.

[9] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern recognition letters* 27, 8 (2006), 861–874.

[10] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 426–437.

[11] Bo Jiang, Zhenyu Zhang, TH Tse, and Tsong Yueh Chen. 2009. How well do test case prioritization techniques support statistical fault localization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1. IEEE, 99–106.

[12] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. 2016. System-level test case prioritization using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 361–368.

[13] Hang Li. 2011. Learning to rank for information retrieval and natural language processing. *Synthesis Lectures on Human Language Technologies* 4, 1 (2011), 1–113.

[14] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*. 688–698.

[15] Jackson A Prado Lima and Silvia R Vergilio. 2020. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020), 106268.

[16] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[17] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 540–543.

[18] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 233–242.

[19] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948.

[20] David Saff and Michael D Ernst. 2004. An experimental evaluation of continuous testing during development. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 76–85.

[21] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2018. Reinforcement learning for automatic test case prioritization and selection in continuous integration. *arXiv preprint arXiv:1811.04122* (2018).

[22] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 97–106.

[23] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

[24] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster fault finding at Google using multi objective regression test optimisation. In *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary.*