CS-TR-2018-001: An Empirical Study of Continuous Integration Failures in TravisTorrent

Foyzul Hassan, Xiaoyin Wang
The University of Texas at San Antonio
Email: {foyzul.hassan, xiaoyin.wang}@utsa.edu

July 3, 2018

Abstract

Continuous Integration(CI) is a widely used development practice where developers integrate their work after submitting code changes at central repository. CI servers usually monitor central repository for code change submission and automatically build software with changed code, perform unit testing, integration testing and provide test summary report. If build or test fails developers fix those issues and submit the code changes. Continuous submission of code modification by developers and build latency time creates stalls at CI server build pipeline and hence developers have to wait long time to get build outcome. In this paper, we categorize CI failures from TravisTorrent data set and studied the frequency of different failures. Our study on 1,187 CI failures from TravisTorrent data set shows that 829 of the CI failures are test failures, compared with 252 build script errors, 49 JavaDoc errors, and 67 stylechecking errors. It also reveals that 10.8% of CI-failure repair commits contains only build script revisions, and 25.6% CI-failure repair commits involve at least one build script revision.Furthermore, we proposed build prediction model that uses TravisTorrent data set with build error log clustering and AST level code change modification data to predict whether a build will be successful or not without attempting actual build so that developer can get early build outcome result. With the proposed model we can predict build outcome with an average F-Measure over 87% on all three build systems (Ant, Maven, Gradle) under the cross-project prediction scenario.

1 Introduction

Software development process and speed have changed drastically over the years with more distributed development teams. With the requirement growth, software development teams need to adopt novel practices with evolving social coding and process automation platforms. These practices allow distributed teams to work closely and help to increase productivity. Projects that are more popular, are subject to higher delivery pressure and more frequent release cycle, and thus require more rapid development, testing, integration etc. Broken integration will slow down the release process, and as a result we need to have more centralized integration process.

To ensure the system is functioning with every code changes made by the developers, one key innovative idea is Continuous Integration(CI) [19] and this process has been adopted by many organizations for faster integration. A typical CI system attempts to automate the whole software build process from compilation to test case execution. For CI systems, dedicated infrastructure with different build systems such as Make, Ant, Maven, and Gradle are used to automate the building process. Despite the growing interest in CI, build failures in respect to CI process is an under-explored area. Even though CI is used for continuous development, the integration process might be delayed for long build chains, build error fixes, and frequent code changes in version control system. According to analysis on TravisTorrent [3] data, the median build time for Java project is over 900 seconds and the median length of continuous build failure sequences is 4. Thus when multiple developers are committing their changes concurrently, their commits may be piled up in the building queue, and they may need to wait for a long time to get the build feedback. They also have the option to continue working on their copy, but they will be at the risk of rolling back their changes if their original commit fails the integration.

Therefore, it is desirable to have a recommendation system that predicts the build feedback of a code commit and thus gives developers more confidence to continue their work and reduce the chance of rolling back. In this paper, we analyzed software build execution time, commit and consecutive build status change to study the necessity and possibility of a change-aware build prediction model. Furthermore, we propose a recommendation system to predict build outcome based on the TravisTorrent data and also the code change information in the code commit such as import statement changes, method signature changes to train the build prediction model.

To evaluate our approach, we conducted an experiment with TravisTorrent data set. We focused on Java projects using Ant, Maven and Gradle build systems because they are supported by the TravisTorrent data set. Our evaluation results show

that our model can can achieve an average F-Measure of 87% on all three build system for cross-project build-outcome prediction, which is a very challenging but more realistic usage scenario.

Our paper makes the following main contributions.

- A statistic study of CI build status and build time, and failures on the TravisTorrent data set.
- A build-outcome prediction model based on combined features of the build-instance meta data and code difference information of the commit.
- A large-scale evaluation of our project with both scenarios of cross-validation and cross-project prediction on the TravisTorrent data set with more than 250,000 build instances.

The rest of this paper is organized as follows. We first introduce some existing research efforts in Section 2. Then, we present an empirical study on the building status of the data set in Section 3. Our prediction model is presented in Section 4, followed by our evaluation in Section 5. Before we conclude in Section 7, we discuss the threats to the validity of our evaluation in Section 6.

2 Related Work

Over decades, researchers have been worked on defect prediction models [26] to guide software testing [12, 20] and quality assurance [18, 21, 16]. These defect prediction models are used to identify early defect prone components and thus reduce development time and also reduce maintenance cost. These models are general and does not consider features specific for software builds. Build co-change prediction models [24] are used to identify when we need to change build configuration files to avoid possible build failures, but does not predict build outcome. Wolf et al. [23] and Irwin et al. [11] utilized social network analysis and Socio-technical analysis to predict possible build failures in a project. Bird and Zimmerman [4] discussed software build error prediction and potential approaches, but they did not provide detailed techniques and evaluation. Finalay et al. [7] proposed an software-metrics-based approach to predict build output, but it requires design and history features so that cannot be easily applied to unprepared projects. These works considers isolated workstation environment for prediction model, but CI environment has different work-flow and can suffer with different latency as discussed in different studies [10] [2]. Recently, we [9] studied the reasons of build failures in a large number of projects. Ansong and Ming proposed build outcome prediction model [13] for

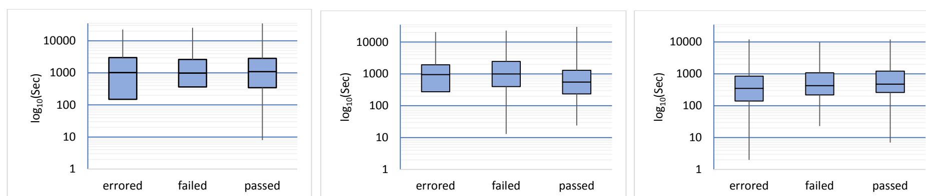

fig1: Ant Build Execution Time

fig2: Maven Build Execution Time

fig3: Gradle Build Execution Time

Figure 1: Ant, Maven and Gradle Build Execution Time Statistics

predicting build fail in CI environment based on commit information of current and last push with statistical information. Our approach uses build commit information, build error type and code change metric to predict build outcome prediction in CI environment with better performance than previous work [13].

3 DATASET AND CHARACTERISTICS

In this paper we studied TravisTorrent data set [3] on Oct 27,2016. This data set includes 402 Java projects with data for 256,055 build instances. Among this 256,055 build instances, the Ant based build system is used for 104,417 cases, while Maven and Gradle based build systems are used for 104,876 and 44,056 instances, respectively. For better analysis and build prediction model, we also used raw build logs of corresponding data set from TravisTorrent ftp server.

3.1 How Much Time Required for Building?

In the TravisTorrent data set, there are four types of build status: passed, failed, errored and canceled. Canceled build status denotes that build process is interrupted while build in progress. So, we simply ignore this status in our study. Then, we considered passed build status as a successful build, while failed and errored build status both are considered as failed build. For each category of build status, we considered three most popular build systems: Ant, Maven and Gradle and performed analysis on build execution time for Ant, Maven and Gradle build system with different build status. As shown in Figure 1, the median build execution time for errored status with Ant, Maven and Gradle build tool are 1,019, 955 and 352 seconds respectively. For failed build status, the median execution time for Ant, Maven and Gradle are 981, 998, and 426 seconds, respectively. Passed build execution time for Ant, Maven and Gradle are 1,090, 558 and 477 second respectively. The maximum build execution time for each build status for each build tool are much higher than median execution time. For example, Ant and Maven tool

Table 1: Manual Categorization of CI Build Failures

Build Tool	Maven	Gradle	Total
Test Failures	244	585	829
Build Script Errors	100	152	252
Javadoc Errors	17	32	49
StyleCheck Errors	27	40	67
Total	388	799	1187

with errored and failed status takes over 20,000 seconds, while for Gradle build tool it takes over 9,000 seconds for errored and failed build status. So, developers typically need to wait for a long time to get the build feedback and it creates stall in the CI process which is also supported by other studies [25].

3.2 What are Major Types of CI Failures?

In our study, we further studied the major types of CI failures in the TravisTorrent data set. To make sure we understand the root cause of CI failures, instead of extracting keywords from build logs, we manually inspected all 1,187 CI Failures within three months from April 2017 to June 2017. Since Ant is getting less popular, we consider only projects using Maven and Gradle as their build systems. The break down of CI Failures are presented in Table 1. We found that they can be categorized into four classes: test failures, build script errors, Javadoc errors, and stylechecking errors.

3.3 How are CI Failures Fixed?

We further studied how the CI Failures are fixed by developers. In particular, we count how different files are being revised in the fixing commits of developers, and the result is shown in Table 2. In particular, we consider as a repair commit a code commit that causes the build status to change from "Failure" to "Success". From the table, we can see that 25.6% CI-failure repair commits involve at least one build script revision

3.4 What is the Time Interval in Between Two Commit?

Code commit time interval between two consecutive commits within the same project can tell how often developers need CI feedback. We performed commit interval frequency based on time interval of two subsequent commits of TravisTorrent data set. According to our analysis shown in Figure 2, 19.54% code commits

Table 2: Fixes of CI Build Failures

Build Tool	Maven	Gradle	Total
Source Files Only	3313	1646	4959
Build Scripts Only	501	314	815
Build Scripts & Source Files	577	544	1121
Other	424	219	643
Total	4815	2756	7571

occurs within 500 seconds, 26.74% of commits occurs within 1,000 seconds, and 54.53% of commit occurs within 10,000 seconds. According to Section 3.1, the median build execution time is 500 to 1,000 seconds. So, for about 20% of the code commits, software builds will line up in the building queue and cannot be performed immediately.

3.5 How Often Consecutive Build Status Changes?

We performed analysis on how often build status changes, or for how many consecutive code commits, the build status remains unchanged. Figure 3 shows the statistical result on consecutive build status for errored, failed and passed. For errored and failed build status, build outcome remains unchanged with median of four build instances. For maximum case, 422 consecutive build was errored, while 760 consecutive build was failed. Such consecutive errors and failures imply that developers keep working on their parts in spite of the build failures, or without knowing about the build failures.

4 Overview of Build Prediction Model

The overall architecture of build prediction model is shown in Figure 4. Our approach consists of three parts: data filtering, feature generation, and model generation. In the data filtering phase, we extract useful information from the build logs. Then we generate features from the extracted data and the code change information in the code commit to be predicted. With the features, we train and evaluate our predict model with Weka [8].

4.1 Data Filtering

For our research, we considered Java projects that uses Ant, Maven and Gradle to predict build outcome prediction model. So, filtered Java projects that uses Ant,

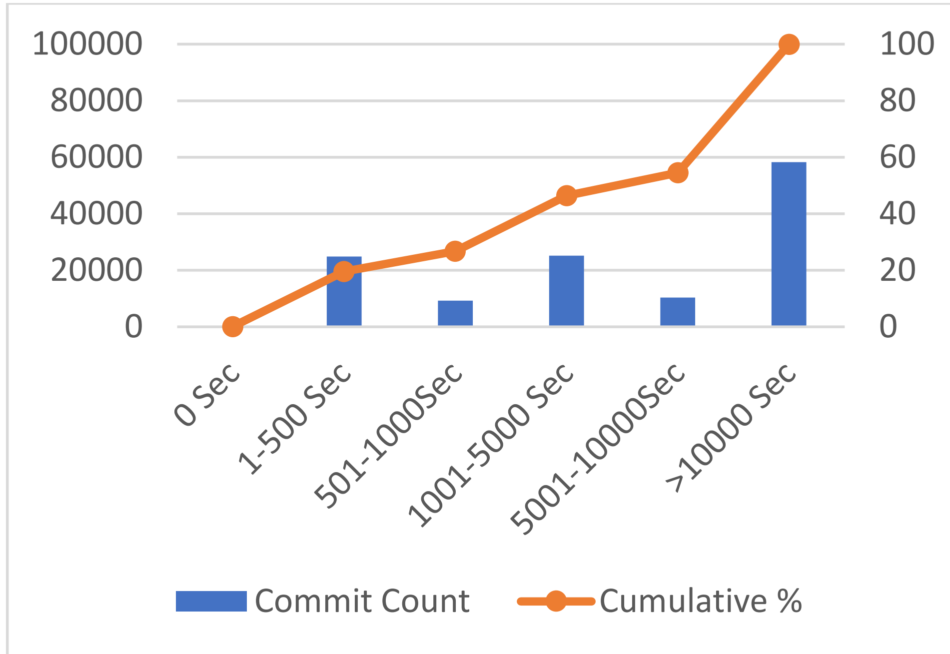

Figure 2: Commit Frequency Time Distribution

Figure 3: Build Status Statistics on Consecutive Build Sequence

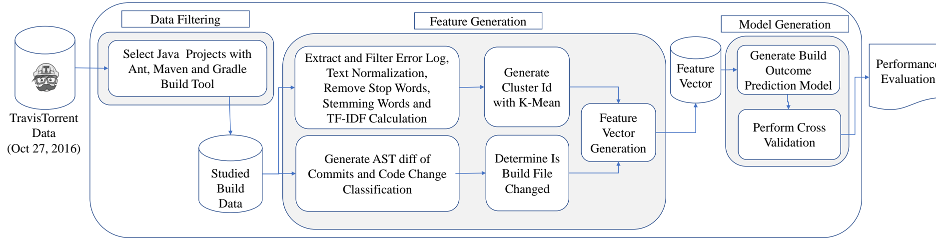

Figure 4: Overview of Build Prediction Model

Maven and Gradle for our research consideration. Based on the filtered build data, we performed feature generation and model generation to predict build outcome in CI environment.

4.2 Feature Generation

4.2.1 Build Failure Cluster ID Generation

Although successful builds are almost always the same, build failure can happen for different reasons such as compilation issues, dependency issues, or tools version issues. Information about different types of build failures are useful for predicting the successfulness of the following builds. In our approach, we fetched raw build logs from TravisTorrent ftp server for errored and failed build status and performed document clustering to group similar build failures in same group and uses cluster id to denote each failure group. During cluster id generation, we first filter build log text of Ant, Maven and Gradle build logs. Build logs denote different activities such as downloading dependencies, compiling source files, errors if happening during build. We are mainly interested in errors and exceptions in build log file, because they are more relevant to the type of build errors. For passed build instances, we considered them as a single cluster as passed.

Specifically, we developed regular expression based parser for Ant, Maven and Gradle to extract only error and exception log parts. After filtering, we performed text normalization and stop word removal process. Then we performed stemming with popular Porter stemming algorithm [22]. After that, we calculated Term Frequency–Inverse Document Frequency (TF–IDF) [15] for denoting statistical uses of a word in given document(s) to reflect the importance of the word. With TF-IDF value, we performed clustering using popular K–Mean Clustering [17] algorithm. For K–Mean clustering value of K is calculated using formula $\sqrt{n/2}$, where n is the number of build logs. We calculate the cosine similarity of the query's vector and the cluster centroid's vector. Among the generated clusters compilation failures, test failures, and dependency resolve failures are most prominent clusters.

Examples 1 and 2 show build log part of two different project having dependency resolve failure issue are clustered in same group with our approach and denoted by same cluster id.

Example 1 *(gh_project_name: BuildCraft/BuildCraft, tr_build_number: 954)*

```
A problem occurred configuring root project 'BuildCraft'.
> Could not resolve all dependencies for configuration ':
    classpath'.
```

Example 2 *(gh_project_name: MrTJP/ProjectRed, tr_build_number: 453)*

```
Could not resolve all dependencies for configuration ':
    compile'.
> Could not resolve codechicken:CodeChickenLib
    :1.7.10-1.1.1.104.
```

4.2.2 Features from Code Commits

To further consider features from the source code change of the code commit, for each build instance, we extracted the commit hash from TravisTorrent data set. For each commit, we further explore the changes with JGit [1], a Git client for Java, and generated the AST diff of the code commit with GumTreeDiff [6]. Based on the the AST diff of the code commits, we build features: `Import Statement`, `Class Signature`, `Attributes`, `Method Signature` and `Method Body` changes and count the number of changes as feature value. Apart from above Java code change features, we also considered file structure level features, such as how many build script such as pom.xml, build.gradle and build.xml files are changed.

4.3 Model Generation

4.3.1 Feature Selection for Model Generation

We applied Information Gain(IG) attribute evaluation [14] algorithm to select discriminating features from the feature set. For information gain attribute, entropy is a commonly used for characterizing the purity of information. Entropy is is used for IG attribute ranking methods. The entropy measure is considered as a measure of system's unpredictability. The entropy of Y is

$$H(Y) = -\sum_{y \in Y} p(y) log_2(p(y)) \tag{1}$$

Table 3: Features Used for Build Prediction Model

Build Instance	Feature Name	Source of Feature
Previous Build Instance	prev_bl_cluster	Generated
	prev_tr_status	TravisTorrent
	prev_gh_src_churn	TravisTorrent
	prev_gh_test_churn	TravisTorrent
Current Build Instance	gh_team_size	TravisTorrent
	cmt_buildfilechangecount	Generated
	gh_other_files	TravisTorrent
	gh_src_churn	TravisTorrent
	gh_src_files	TravisTorrent
	gh_files_modified	TravisTorrent
	gh_files_deleted	TravisTorrent
	gh_doc_files	TravisTorrent
	cmt_methodbodychangecount	Generated
	cmt_methodchangecount	Generated
	cmt_importchangecount	Generated
	cmt_fieldchangecount	Generated
	day_of_week	Generated
	cmt_classchangecount	Generated
	gh_files_added	TravisTorrent
	gh_test_churn	TravisTorrent

where p(y) is the probability density function for the random variable Y. Information Gain Attribute Evaluation entropy value resides in between 0 to 1. Higher entropy value indicates higher effectiveness of the feature. During feature selection we considered four features of the previous build instance, and all remaining features are from the current build instance. Table 3 shows the list of used features. Besides the existing TravisTorrent features (marked as TravisTorrent in Column 3), we also generated other features marked as Generated. Description of our generated features are provided in Table 4.

4.3.2 Build Prediction Classifier Construction

We construct classifier using the random forest algorithm [5] of Weka implementation. The random forest classifier produces distinct decision trees which are random subset of all model attributes, and we use random forest algorithm because the algorithm has been reported to be most effective in existing software engineering

Table 4: Generated Feature Description

Feature Name	Description of Feature
prev_bl_cluster	Previous Build Cluster ID
cmt_buildfilechangecount	Number of build script file change
cmt_methodbodychangecount	Number of method body change count
cmt_methodchangecount	Number of method signature change
cmt_importchangecount	Number of import statement changes
cmt_fieldchangecount	Number of class attribute change
day_of_week	Day of week of the first commit for the build
cmt_classchangecount	Number of class changed

research efforts [24]. The classifier calculates a classification decision for each of the trees and then aggregates the partial results to a total classification result.

5 Evaluation and Result Study

In this section, we evaluated our proposed model on Java projects that uses Ant, Maven or Gradle from TravisTorrent data set [1]. During evaluation we tried to answer following research questions.

- **RQ1:** To what extent our build prediction model can successfully predict build outcome for Cross Validation and Cross Project prediction model?
- **RQ2:** Which feature attributes of our models are important for build outcome prediction in CI environment?

To address **RQ1**, we evaluated our build prediction model for Ant, Maven and Gradle build systems. As build errors of Ant, Maven and Gradle are different and build-log formats are different for these tools, we construct separate model for Ant,

[1] The traing, testing and SQL dump files used in our evaluation are available at https://drive.google.com/drive/folders/0B8nZRb4sCPS_RUF4SWtFa0tIQUU?usp=sharing

Table 5: Performance Evaluation of Build Prediction Model

Build Tool	Precision	Recall	F-Measure	ROC Area	Class/Type
Ant	0.948	0.948	0.948	0.975	Pass
	0.923	0.924	0.923	0.975	Fail
	0.938	0.938	0.938	0.975	Weighted Avg.
Maven	0.960	0.963	0.961	0.950	Pass
	0.838	0.825	0.831	0.950	Fail
	0.937	0.937	0.937	0.950	Weighted Avg.
Gradle	0.945	0.956	0.951	0.936	Pass
	0.830	0.794	0.812	0.936	Fail
	0.921	0.922	0.921	0.936	Weighted Avg.

Table 6: Confusion Matrix of Build Prediction Model

Build Tool	Actual: Pass	Actual: Fail	
Ant	51766	2864	Predicted: Pass
	2826	34121	Predicted: Fail
Maven	70071	2692	Predicted: Pass
	2952	13907	Predicted: Fail
Gradle	29330	1338	Predicted: Pass
	1694	6535	Predicted: Fail

Maven and Gradle. We performed 8-Fold Cross Validation with feature set mentioned at Table 3 with random forest learning algorithm for Ant, Maven and Gradle build prediction. Table 5 shows build outcome prediction model performance and Table 6 shows confusion matrix of prediction model to give better idea of what our classification model is getting right for pass and fail build status.

According to Table 5 for both Ant and Maven average Precision, Recall, F-Measure and ROC Area are above 0.93. While for Gradle, average precision, recall and F-Measure is above 0.92. Confusion Matrix provided at Table 6 gives idea about successful prediction rate of our model for both pass and fail classes. According to the table, Ant build tool successfulness for pass class is 94% and fail class is 92%. For Maven successfulness for pass class is 95% and fail class is 83%, while for Gradle successfulness for pass class is 94% and fail class is 83%. For all build tool, fail class instances are less than pass class instances in training and testing. As result, performance for fail class prediction rate is less than the pass class prediction.

Table 7: Cross Project Performance Evaluation of Build Prediction Model

Build Tool	Precision	Recall	F-Measure	ROC Area	Class/Type
Ant	0.920	0.918	0.919	0.938	Pass
	0.907	0.909	0.908	0.938	Fail
	0.914	0.914	0.914	0.938	Weighted Avg.
Maven	0.929	0.949	0.939	0.927	Pass
	0.853	0.802	0.827	0.927	Fail
	0.909	0.910	0.909	0.927	Weighted Avg.
Gradle	0.908	0.919	0.913	0.873	Pass
	0.777	0.752	0.764	0.873	Fail
	0.872	0.873	0.872	0.873	Weighted Avg.

Table 8: Confusion Matrix of Cross Project Build Prediction Model

Build Tool	Actual: Pass	Actual: Fail	
Ant	4186	376	Predicted: Pass
	366	3668	Predicted: Fail
Maven	10658	569	Predicted: Pass
	811	3290	Predicted: Fail
Gradle	3069	272	Predicted: Pass
	312	946	Predicted: Fail

Apart from 8-Fold Cross validation, we also performed Cross Project validation, in which for each build system (Ant, Maven, and Gradle) 80 percent of the projects (alphabetically higher ranked) are used as training sets and the remaining 20 percent of the projects are used as testing sets. Table 7 shows performance evaluation of cross-project validation and Table 8 shows confusion matrix of cross-project evaluation.

For Cross Project evaluation, the effectiveness of build prediction models drop a bit, but for Ant and Maven it can still predict build outcome with over 0.90 F-Measure. For Gradle, our build prediction model can predict build outcome with over 0.87 F-Measure. Successfulness for pass and fail class for Ant build is 91% and 90% respectively. For Maven, correctly predicted pass class is 92% and fail class is 85%. While for Gradle pass class successful perdition rate is 90% and fail class successful prediction rate is 77%.

To answer **RQ2**,during feature selection we applied Information Gain Attribute Evaluation on Ant, Maven and Gradle data set and select those attributes having

Table 9: InfoGainAttributeEval Entropy for Ant, Maven, Gradle and Average for Top Ten Features

Feature Name	Ant	Maven	Gradle	Avg
prev_bl_cluster	0.6661	0.3963	0.3811	0.4812
prev_tr_status	0.6444	0.3893	0.3750	0.4696
gh_team_size	0.0403	0.0181	0.0354	0.0313
gh_src_churn	0.0141	0.0050	0.0044	0.0078
prev_gh_src_churn	0.0089	0.0045	0.0043	0.0059
cmt_buildfilechangecount	0.0049	0.0044	0.0083	0.0058
cmt_importchangecount	0.0105	0.0060	0.0007	0.0057
cmt_methodbodychangecount	0.0113	0.0039	0.0016	0.0056
gh_test_churn	0.0083	0.0079	0.0004	0.0056
prev_gh_test_churn	0.0084	0.0074	0.0006	0.0055

average entropy>0.005. Table 9 shows the top ten features features among the used features for our model generation with average entropy from high to low.

Among the feature set, prev_bl_cluster and prev_tr_status are most prominent. prev_bl_cluster feature denotes build failure category type. For example, build failure category type can be dependency missing and that case build it might requires to change source code import statement update or build configuration file update. So, prev_bl_cluster feature gives high entropy value to other feature such cmt_importchangecount,cmt_importchangecount etc. and considered as most prominent feature. Apart from that, prev_tr_status feature is also considered as important feature. In most cases if previous build status is passed, then next build has higher probability of passing. Similarly, build failure has long chain of continuous build failure.

6 Threats To Validity

Regarding the validity of our experiment, we identify the following threats to the internal, external and construct validity.

Internal Validity. One threat to internal validity is related to training and test set selection. We tried to mitigate the issue with cross-validation and cross-project validation. Even after that there might be issue pass and fail class imbalance due to nature of build sequence pattern. Others threat to internal validity might be during feature selection. During feature selection, we tried to select feature from TravisTorrent data set and also we generated other features related to code change and build error type. There might have other attributes that could be helpful for

build outcome prediction.

External Validity. Our experimental results might have concerns of generalizability, since we performed the experiments with TravisTorrent projects those adopted TravisCI for continuous integration. However, further experiments with commercial systems, projects with different programming languages and systems using other build tools instead of Ant, Maven and Gradle can give generalized result.

Construct Validity. To generate prediction model, we grouped similar failure cases in same group using clustering algorithm. Due to different text pattern of build logs, our approach might fail to group similar bugs in same group. We tried to mitigate the issue using regular expression based filtering of build log text. Apart from that during code change classification we used GumTree. GumTree has been widely used for AST level code diff. But due to versatile pattern of code change, there can have issues that we might failed to extract appropriate code change type.

7 Conclusion and Future Work

Although several approaches have been developed for predicting build co-changes or component failure, we propose the scalable approach for predicting build outcome in CI environment with evaluation on large scale data. Our evaluation shows that our approach predicts build outcome with over 87 percent F-Measure for all build systems in CI environment. Our approach will help developers to get early build outcome without making actual build. This model even can also be helpful for reducing CI computation resources. For current build prediction model we considered code change of source file, not build configuration file. In future, we are planning to use build configuration change type as feature for build outcome prediction model. Apart from that, different learning algorithms can be used for better accuracy.

Acknowledgments The authors are supported in part by NSF Grant CCF-1464425.

References

[1] Jgit. `https://eclipse.org/jgit/`. Accessed: 2017.

[2] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. Technical report, PeerJ Preprints, 2016.

[3] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *MSR*, 2017.

[4] C. Bird and T. Zimmermann. Predicting software build errors, Feb. 20 2014. US Patent App. 13/589,180.

[5] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[6] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ASE*, pages 313–324, 2014.

[7] J. Finlay, R. Pears, and A. M. Connor. Data stream mining for predicting software build outcomes using source code metrics. *Inf. Softw. Technol.*, 56(2):183–198, 2014.

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[9] F. Hassan, S. Mostafa, E. Lam, and X. Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *ESEM*, 2017.

[10] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*, ASE 2016, pages 426–437, New York, NY, USA, 2016. ACM.

[11] I. Kwan, A. Schroter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *TSE*, 37(3):307–324, 2011.

[12] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *QSIC*, pages 127–132. IEEE, 2014.

[13] A. Ni and M. Li. Cost-effective build outcome prediction using cascaded classifiers. In *MSR*, MSR '17, pages 455–458, Piscataway, NJ, USA, 2017. IEEE Press.

[14] J. Novakovic. Using information gain attribute evaluation to classify sonar targets. In *17th Telecommunications forum TELFOR*, pages 1351–1354, 2009.

[15] J. Ramos et al. Using tf-idf to determine word relevance in document queries. In *ICML*, 2003.

[16] Y. Song, X. Wang, T. Xie, L. Zhang, and H. Mei. Jdf: detecting duplicate bug reports in jazz. In *ICSE*, volume 2, pages 315–316. IEEE, 2010.

[17] M. Steinbach, G. Karypis, V. Kumar, et al. A comparison of document clustering techniques. In *KDD workshop on text mining*, volume 400, pages 525–526. Boston, 2000.

[18] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, pages 83–95. ACM, 2015.

[19] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 805–816, New York, NY, USA, 2015. ACM.

[20] X. Wang, L. Zhang, and P. Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *ISSTA*, pages 199–210. ACM, 2015.

[21] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470. IEEE, 2008.

[22] P. Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.

[23] T. Wolf, A. Schröter, D. Damian, and T. H. D. Nguyen. Predicting build failures using social network analysis on developer communication. In *ICSE*, pages 1–11, 2009.

[24] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan. Cross-project build co-change prediction. In *SANER*, pages 311–320, 2015.

[25] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *MSR*, MSR '15, pages 367–371, Piscataway, NJ, USA, 2015. IEEE Press.

[26] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *MSR*, pages 182–191. ACM, 2014.