Predicting build outcomes in continuous integration using textual analysis of source code commits

Citation for the original published paper (version of record):
Al Sabbagh, K., Staron, M., Hebig, R. (2022). Predicting build outcomes in continuous integration using textual analysis of source code
commits. PROMISE 2022 - Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, co-located with ESEC/FSE 2022: 42-51.
http://dx.doi.org/10.1145/3558489.3559070

Predicting Build Outcomes in Continuous Integration using Textual Analysis of Source Code Commits

Khaled Al-Sabbagh*
Miroslaw Staron
Regina Hebig
khaled.al-sabbagh,miroslaw.staron,regina.hebig
Chalmers | University of Gothenburg, Computer Science and Engineering Department
Gothenburg, Sweden

ABSTRACT

Machine learning has been increasingly used to solve various software engineering tasks. One example of its usage is to predict the outcome of builds in continuous integration, where a classifier is built to predict whether new code commits will successfully compile. The aim of this study is to investigate the effectiveness of fifteen software metrics in building a classifier for build outcome prediction. Particularly, we implemented an experiment wherein we compared the effectiveness of a line-level metric and fourteen other traditional software metrics on 49,040 build records that belong to 117 Java projects. We achieved an average precision of 91% and recall of 80% when using the line-level metric for training, compared to 90% precision and 76% recall for the next best traditional software metric. In contrast, using file-level metrics was found to yield a higher predictive quality (average MCC for the best software metric= 68%) than the line-level metric (average MCC= 16%) for the failed builds. We conclude that file-level metrics are better predictors of build outcomes for the failed builds, whereas the line-level metric is a slightly better predictor of passed builds.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Continuous Integration, Build Prediction, Textual Analysis

ACM Reference Format:
Khaled Al-Sabbagh, Miroslaw Staron, and Regina Hebig. 2022. Predicting Build Outcomes in Continuous Integration using Textual Analysis of Source Code Commits. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '22), November 17, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3558489.3559070

1 INTRODUCTION

Continuous integration (CI) is a modern software engineering practice in which developers integrate their code into a shared repository to enable swift detection of quality issues and bugs before releasing new features to end users [19].

A CI system typically attempts to launch a build job multiple times a day, either for each new commit submitted to the version control system or at set time intervals during the day [4]. The goal of these jobs is to notify software engineers about faults in the source code as quickly as possible in order to quickly fix them. A typical build server runs tools such as compilers and static analyzers to detect styling and quality related problems in the code that get reported to developers. Failures produced by any of these tools result in a build failure.

The completion of build jobs in a fast manner directly affects the productivity of programmers [14], as they might get distracted by other tasks while waiting for the build job to finish. As a consequence, the number of code changes committed by developers during a day will be reduced. For this reason, keeping a high pace of the build job, and understanding the root cause of build failures is key to improve the development productivity. In fact, a previous analysis on the TravisTorrent database image, created on February 8, 2017, showed that the median time to build Java projects took over 900 seconds (15 minutes) [8]. This means that developers will incur, on average, a time latency of 15 minutes before receiving feedback about their committed code from the CI environment. Therefore, reducing the time-feedback to developers is necessary to allow them to immediately start working on new development tasks with confidence that their previously committed code will pass the build phase.

To address the problem of time latency and reduced development productivity in CI, several researchers utilized machine learning (ML) models to predict the outcome of build jobs using a diversity of product and process software metrics, such as code churn size, number of commits, team size etc as features for characterizing build outcome (failed/passed). For example, Hassan and Zhang [7] mined a diversity of product and process metrics in historical projects, such as the number of modified subsystems and certification results of previous build for constructing an ML model for build prediction. Their results indicate that training a decision tree classifier on such information can yield to a correct prediction for 95% of passing builds and 69% of failing builds. Xia and Li [16] evaluated the use of nine classifiers on 20 software metrics for 126 open source projects. Their results show that using the examined metrics result in an

F1-score higher than 0.7 for 21 of build outcomes. Thus, product and process metrics have shown promising results when it comes to prediction of build outcomes. In this study, we refer to these metrics by using the term traditional software metrics (TSM).

Despite these promising results of TSM-based approaches, they can only provide indications about which parts of the system, e.g. what file, causes the build to fail. However, they cannot locate the source of the failure, e.g. by indicate which lines of code might potentially cause build failures. This research aims at filling this gap. Using a textual analysis (TA) approach, we measure the frequency count of token appearances in the source code (e.g., if and while) on a line of code level. We use the term token frequency (TF) metric to refer to the measurements produced by the TA approach. However, in the context of build outcome prediction, it is unclear whether a prediction of build failure made using a line-level metric, such as TF, can be as precise and good as a prediction made using TSM, which can use per file information.

Therefore, in this study we set off to examine the effectiveness of the TF metric by empirically comparing its effectiveness against a set of TSM in build outcome prediction for 117 Java open source projects. We record the precision, recall, F1-score, and Mathews Correlation Coefficient (MCC) measures attained after training a classifier on each metric respectively. More concretely, we design and implement a controlled experiment wherein fourteen different TSM metrics extracted from the TravisTorrent [5] data-set, created on February 8, 2017, and the token frequency metric are examined for effectiveness across 49,040 builds. In our study, we address the following research question:

> *How effective is the token frequency metric in comparison with traditional software metrics for predicting build outcomes in CI?*

The specific contributions of this paper are as follows:

- we empirically investigate the effectiveness of a line-level metric in learning build outcomes in CI, and compare its effectiveness with a diversity of traditional software metrics using 117 Java based open source projects.
- we found that using TF for training a classifier slightly outperforms the effectiveness of file-level TSMs in predicting passing builds, with a small effect size difference.
- we found that using file-level metrics is more effective than TF in predicting build failures.
- we complement the TravisTorrent data-set from 2017 with a new data-set that contains TSM and TF metrics for historical code changes made in 117 Java based projects [1].

The remainder of this paper is organized as follows. Section 2 provides an overview of related work that propose approaches for CI build prediction. In Section 3 we present the experimental design and operations carried out in this study. Section 4 presents the results of our study. Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper and outlines future work.

2 RELATED WORK

In this section, we present related work on build outcome prediction and reasons of build failures in CI.

[1]https://doi.org/10.5281/zenodo.6784987

2.1 Software Metrics for Build Prediction

Several studies have proposed approaches for modelling the relationship between build statuses (passed/failed) and software metrics [8, 10, 16]. Ni and Li [10] adopted cascaded classifiers to predict build outcomes using 18 software metrics to characterize historical build jobs for 532 Java and Ruby projects. The results showed that using 'Historical Statistic' metrics are the most useful features in predicting the build outcome with an accuracy rate of 75.3%. Hassan and Wang [8] employed a random forest classifier for predicting build outcomes using features derived from error logs in historical build records. The results of their work showed an average F-score rate of 87% in the prediction of build outcomes. Another example is the work conducted by Xia and Li [16], where the authors evaluated the performance of nine different classifiers using traditional software metrics in build predictions. Their results showed that a Decision tree, gradient boosting and random forest classifiers outperform the other algorithms in F1-score, achieving a 17% more F-score on average. With these classifiers, build outcomes for a quarter of the analyzed projects can be predicted with F1-score over 60%. In another empirical study conducted by Xia et al. [17], the authors evaluated the performance of six classifiers for build outcome prediction. The results of their study revealed that a Decision Tree classifier performs the best in comparison with the other five classifiers with a score of 17% for F1-score on average.

Despite the promising results that the majority of these studies achieved, non of them has investigated the effectiveness of metrics that operate on a line of code level for build prediction. In this paper, we characterize historical changes of source code on a line-level of abstraction and analyze its effectiveness in predicting build failures.

2.2 Reasons of Build Failure in CI

Over the recent years, studies on identifying factors that result in build failures are increasingly gaining more attention by researchers [4, 9, 13]. Rausch et al. [13] investigated factors that result in build failures. The findings drawn from the analyses of historical build logs suggest that failing integration tests, code quality, and compilation errors are the most common factors that lead to build failures. Luo et al. [9] used the TravisTorrent data-set to investigate factors that cause build failures. They found that in our study, the number of commits in a build (git_num_all_built_commits) is the most import factor that has significant impact on the build result. Beller et al.[4] conducted an in-depth quantifiable study using TravisTorrent data-set to investigate the effect of testing on build failures. The results of their work concluded that testing is the most important factors that leads to build failure. Moreover, the programming language has a strong impact on the number of executed tests, the time they take to execute, and their proneness to fail. In this paper, we expand on these empirical studies by examining the effectiveness of a new metric (token frequency) which can potentially identify the root cause of build failures in the source code.

3 EXPERIMENT DESIGN AND OPERATIONS

This section describes the experiment design, the data-sets, and the operations carried out to implement this experiment.

3.1 Data Collection and Preprocessing

TravisTorrent is a synthesized open-source data-set that consists of 2,640,825 build job records belonging to 1,300 projects (402

Java projects and 898 Ruby projects) [5]. Every build job record in the data-set synthesizes information from three data sources: The project's git repository, data extracted from GitHub, and data from Travis's API. In total, the data-set provides 55 software metric values for each historical build.

3.1.1 *Traditional Software Metrics.*

In this study, we utilized the BigQuery interface for the TravisTorrent data-set, created on February 8, 2017, to mine historical build records for fourteen traditional software metrics. Table 3 provides a brief summary of each metric. We chose to only examine the effectiveness of these metrics as they characterize changes made to the source code (product specific) and the process, whereas the remaining metrics in TravisTorrent characterize test related aspects (e.g., tests added and tests deleted). Further all the selected traditional software metrics were previously examined in different build prediction studies such as [8, 10].

Since the goal of this study is to evaluate the effectiveness of different software metrics in learning build outcome (pass/fail), we restrict the sample of collected historical build records and projects to fulfill the following two criteria. First, we filtered out all records whose build status (tr_build) values resolved to errored or canceled, and only kept track of those that resolved to passed/failed. Second, we only queried projects that were written in the Java programming language and included at least one failing/passing build job record. The outcome was a data-set that comprised of 117 Java projects with a total of 49,040 build records. Information about the distribution of the collected build status records and project names are summarized in columns 'Failing Builds' and 'Passing Builds' in Table 1.

3.1.2 *The Token Frequency Metric.*

To instrument the measurement of the token frequency metric, we implemented a TA based tool that follows the procedure introduced in [2]. The procedure enacts three sequential steps that can be summarized as follows:

Step 1 (extraction of code changes): This step involves extracting code changes committed to the development repository of each analyzed project. For each project, we extract modified/added lines of code between pairs of consecutive builds. All extracted lines between each pair are then labeled with the execution outcome (passed/failed) of the newer build. The build execution outcomes are provided in TravisTorrent under the field 'build_status'. Thereafter, we save the extracted lines of code for every project in a 'csv' file for every analyzed project. A total of 117 csv files (one file for each project) were collected and stored locally before being processed in step 2 of the TA procedure[2].

Step 2 (features extraction): The second step utilizes a textual analysis tool [11] to convert the corpus of extracted code changes in step 1 into feature vectors. For each line of code in the collected corpus, the tool:

- creates a vocabulary for all lines of code (using the BoW technique, with a cut-off parameter of how many words should be included[3])
- creates a token for words that fall outside of the frequency defined by the cut-off parameter of the bag of words
- finds a set of predefined keywords in each line,
- checks each word in the line to decide if it should be tokenized or if it is a predefined feature.

[2]https://anonymous.4open.science/r/CIbuilds_TSM_TF-CE19/

[3]BoW is essentially a sequence of tokens, which are descendingly ordered according to frequency. This cut-off parameters controls how many of the most frequently used words are included as features – e.g. 10 means that the 10 most frequently used words become features and the rest are ignored.

The output of this step is a large array of numbers, each representing the the token frequency of a specific feature in the bag of words space of vectors. Table 2 illustrates an exemplary output of the bag of words vectors for a simple code fragment written in the C language. In this study, we chose to use a bi-gram model for representing the feature vectors, as it was previously shown to produce good learning performance in a similar context (e.g., [2]). Notice how the feature values in Table 2 correspond to the frequency counts of each token that appears in every line of code in the code example.

Step 3 (training a classifier): Finally, the extracted set of feature vectors from step 2 are fed into an ML model for learning how to classify new lines of code as triggering to CI builds pass/failure. The result of applying the TA technique on the collected projects resulted in extracting historical code changes made to every collected project, as summarized in Table 1 under the 'Lines' column. The distribution of classes assigned to the extracted lines is specified under the columns 'Failing lines' and 'Passing lines' in Table 1.

3.2 Independent Variables

In this study, software metric is the only independent variable (treatment) examined for effectiveness on the performance of a build prediction model. A total of 15 variations (software metrics) to the independent variable were examined, as summarized in Table 3. Detailed description about metrics 1 to 14 can be found in the TravisTorrent database [5], whereas metric fifteen (token frequency) is a measure of the frequency count of code tokens in the analyzed programs using textual analysis.

3.3 Evaluation Metrics

We chose four state-of-the-art metrics to evaluate the performance of a classifier for build outcome prediction that we train on the TSM and TF metrics respectively. The four metrics are precision, recall, F1-score, and Matthews Correlation Coefficient.

While precision is the proportion of correctly identified passing builds, recall is the proportion of relevant builds that were identified as such. Having both precision and recall high ensures the detection of larger amount of passing builds and the reduction of false alarms about failing builds.

The F1-score indicates whether the predictive model is performing well in identifying builds that are actually passing (high precision) and generating little false alarms about failing builds (high recall). One drawback in using the F1-score metric is the fact that it only accounts for three elements in the confusion matrix (true positives, false positives, and false negatives), which might lead to misleading conclusions if the distribution of the binary classes in the training data is imbalanced [18].

To mitigate these drawbacks that suffice in F1-score, we decided to measure the model's MCC, which takes into account the four elements in the confusion matrix [18]. In the context of build outcome prediction, a high MCC indicates that the predictions obtained by the model are good in both classes (passing and failing builds), as MCC takes the four elements of the confusion matrix into account. Thus, MCC considers what share of the elements (builds or lines)

Table 1: Distribution of Build Outcomes and Lines of Code Changes in the Analyzed Projects

Id	Project	Builds	Failing builds	Passing builds	lines extracted	Failing lines	passing lines
1	OpenRefine	192	14	178	3684	332	3352
2	psi-probe	200	4	196	50471	100	50371
3	error-prone	152	3	149	91072	26	91046
4	u2020	245	8	237	4404	135	4269
5	metrics	279	23	256	8504	1667	6837
6	rewrite	184	72	112	13399	3693	9706
7	checkstyle	1368	30	1338	93435	357	93078
8	ProjectRed	268	66	202	1075	385	690
9	brightspot-cms	548	62	486	8737	2753	5984
10	assertj-android	118	36	82	14267	3349	10918
11	LittleProxy	287	42	245	7806	916	6890
12	blueprints	432	127	305	41217	15106	26111
13	cassandra-reaper	262	20	242	7688	1004	6684
14	restlet-framework-java	436	277	159	109038	48701	60337
15	nodeclipse-1	238	13	225	21274	24	21250
16	rultor	1156	275	881	33023	10185	22838
17	jmonkeyengine	714	9	705	69466	782	68684
18	pdfsam	336	91	245	108882	20235	88647
19	robospice	74	29	45	13307	7575	5732
20	pushy	333	21	312	4281	9	4272
21	parceler	227	4	223	15645	232	15413
22	dynjs	320	20	300	32833	899	31934
23	mybatis-3	471	15	456	94630	572	94058
24	HikariCP	326	17	309	29153	3490	25663
25	thredds	333	100	233	24625	6308	18317
26	maven-git-commit-id-plugin	201	31	170	14030	1499	12531
27	dagger	302	24	278	3205	105	3100
28	jade4j	207	11	196	15059	265	14794
29	jsonld-java	196	13	183	32630	76	32554
30	webcam-capture	342	22	320	32386	227	32159
31	jInstagram	219	7	212	18103	242	17861
32	spring-cloud-config	251	22	229	22734	1206	21528
33	gpslogger	265	36	229	14238	918	13320
34	jcabi-http	221	34	187	4329	596	3733
35	p6spy	333	100	233	13584	5920	7664
36	htm.java	442	4	438	49931	413	49518
37	go-lang-idea-plugin	780	81	699	31476	1212	30264
38	Singularity	152	36	116	7302	1861	5441
39	android	671	46	625	13857	3885	9972
40	jcabi-github	502	146	356	17052	5765	11287
41	sms-backup-plus	248	20	228	14921	195	14726
42	truth	96	18	78	17907	1260	16647
43	joda-time	186	5	181	18153	34	18119
44	logback	183	49	134	66311	1775	64536
45	mockito	320	56	264	66687	2774	63913
46	Hystrix	508	202	306	38633	16536	22097
47	blueflood	744	80	664	39209	4160	35049
48	java-design-patterns	630	5	625	69967	51	69916
49	DDT	183	62	121	55702	10375	45327
50	dropwizard	1048	64	984	48830	1070	47760
51	nokogiri	439	117	322	23572	9520	14052
52	android-maven-plugin	224	141	83	74259	4096	70163
53	jcabi-aspects	304	34	270	5354	858	4496
54	intellij-elixir	107	2	105	237184	952	236232
55	jsonschema2pojo	294	1	293	12985	3	12982
56	lorsource	1470	58	1412	31970	656	31314
57	analytics-android	206	17	189	6896	490	6406
58	storm	65	36	29	28317	16622	11695
59	basex	322	40	282	56481	1947	54534
60	spark	249	13	236	5507	83	5424
61	picard	284	11	273	10306	378	9928
62	hivemall	173	17	156	20219	159	20060
63	seyren	281	14	267	7141	136	7005
64	lenskit	274	22	252	54601	808	53793
65	springside4	226	57	169	19121	6279	12842
66	onebusaway-android	187	9	178	19743	63	19680
67	rxjava-jdbc	192	2	190	7847	17	7830
68	core	516	6	510	75569	375	75194
69	selendroid	445	47	398	58650	17944	40706
70	nutz	924	367	557	57793	20243	37550
71	jphp	300	34	266	142552	18063	124489
72	owner	387	7	380	20248	240	20008
73	twilio-java	221	8	213	28005	5547	22458
74	restlet-framework-java	436	277	159	109038	48701	60337
75	azkaban	176	8	168	56976	6026	50950
76	nodeclipse-1	238	13	225	21274	24	21250
77	idea-gitignore	187	45	142	26477	4148	22329
78	keywhiz	240	2	238	12128	5	12123
79	jsprit	210	4	206	20950	118	20832
80	stubby4j	571	144	427	36510	12346	24164
81	qulice	413	33	380	10885	243	10642
82	jinjava	227	3	224	11092	8	11084
83	auto	251	34	217	11912	126	11786
84	xtreemfs	272	41	231	50048	2097	47951
85	jmxtrans	400	22	378	7752	173	7579
86	less4j	647	71	576	72745	7426	65319
87	cas-addons	229	7	222	7022	63	6959
88	goclipse	228	20	208	66453	462	65991
89	ccw	331	142	189	13859	3678	10181
90	unirest-java	301	17	284	3558	225	3333
91	waffle	203	23	180	19207	160	19047
92	MozStumbler	517	12	505	7707	15	7692
93	HearthSim	234	11	223	59681	347	59334
94	rexster	324	23	301	34909	464	34445
95	retrofit	747	5	742	17656	335	17321
96	DSpace	1242	43	1199	77447	2099	75348
97	structr	740	252	488	105605	58803	46802
98	airlift	253	123	130	21916	13609	8307
99	traccar	1324	24	1300	67666	138	67528
100	querydsl	1153	194	959	33926	809	33117
101	yobi	24	2	22	90790	3531	87259
102	openwayback	229	29	200	10976	30	10946
103	cloudify	4137	717	3420	287810	50339	237471
104	play-authenticate	178	27	151	4823	295	4528
105	RoaringBitmap	247	21	226	38994	685	38309
106	jPOS	285	10	275	36033	148	35885
107	javaslang	722	8	714	384967	3997	380970
108	frontend-maven-plugin	273	27	246	2598	106	2492
109	jodd	439	23	416	141987	1363	140624
110	quickml	222	43	179	13670	421	13249
111	okhttp	1341	335	1006	64755	15756	48999
112	bnd	459	24	435	31434	3355	28079
113	AcDisplay	371	187	184	31453	17569	13884
114	jedis	427	61	366	36361	878	35483
115	Hydra	210	35	175	6662	525	6137
116	storio	192	11	181	13058	747	12311
117	Jest	370	71	299	22414	2460	19954

Table 2: Output From the Feature Vectors Using Bag of Words

Filename	Path	`if`	`int`	`a`	`Aa`	Content
firstFile.c	c:/folder	1	0	3	2	`if(condition == true) printf("Hello World");`
firstFile.c	c:/folder	0	0	2	0	`printf("\n");`
secondFile.c	c:/folder	0	1	1	0	`int i = 10;`

in the failing class are correctly identified as failing. If the share is low then MCC is worse than if the share is high.

3.4 Experimental Hypotheses

We hypothesize that using token frequency features for constructing a predictive model is more effective in learning build prediction than traditional software metrics. The hypotheses are based on

Table 3: Descriptions of The Examined Software Metrics

Id	Metric	Description
1	gh_num_commits_in_push	Number of commits in the push that started the build
2	git_prev_commit_resolution_status	String, "merge found" if this build is a merge otherwise "build found"
3	gh_team_size	Size of the team contributing to this project within 3 months of last commit
4	git_num_all_built_commits	Number of all commits in this build
5	gh_num_commit_comments	The number of comments on git commits on GitHub
6	git_diff_src_churn	How much (lines) production code changed by the new commits in this build
7	gh_diff_files_added	Number of files added by the new commits in this build
8	gh_diff_files_deleted	Number of files deleted by the new commits in this build
9	gh_diff_files_modified	Number of files modified by the new commits in this build
10	gh_diff_src_files	Number of production files in the new commits in this build
11	gh_diff_doc_files	Number of documentation files in the new commits in this build
12	gh_diff_other_files	Number of remaining files which are neither production code nor documentation in the new commits in this build
13	gh_num_commits_on_files_touched	Number of unique commits on the files included in this build within 3 months of last commit
14	gh_sloc	Number of executable production source lines of code, in the entire repository
15	token frequency	The frequency count of code tokens in the analyzed source code.

the assumption that build failures are triggered when faults in the code base are introduced. Accordingly, four hypotheses are defined and tested for statistical significance. The hypotheses are formally defined as follows:

- H_{0p}: *The mean precision is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1p} = \mu_{2p} \tag{1}$$

.

- H_{0r}: *The mean recall is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1r} = \mu_{2r} \tag{2}$$

- H_{0f}: *The mean F1-score is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1f} = \mu_{2f} \tag{3}$$

- H_{0mcc}: *The mean MCC is the same for a model trained on token frequency and each traditional software metrics.*

$$\mu_{1mcc} = \mu_{2mcc} \tag{4}$$

3.5 Data Analysis Methods

To decide whether to run a parametric or non-parametric statistical test for analysis, we begin the analysis by running a normality test on the distribution of the four evaluation metrics under the 15 treatment levels. We chose to use the Shapiro Wilk test to evaluate the normality of the distributions. Based on the normality test results, we decided to run the Kruskal-Wallis (a non-parametric test) for comparing the precision, recall, F1-score, and MCC values between the different treatment levels.

While the Kruskal-Wallis statistical test is used to determine statistical significance between the treatment levels and the evaluation metrics (i.e., if the treatment has an effect on precision, recall, F1-score, and MCC), they do not quantify the amount of difference between the groups. Hence, we decided to complement the analysis by calculating the effect size between the precision, recall, F1, and MCC scores attained when using the TF and the next best traditional software metric. For this purpose, we used the 'effsize' library available in R-studio (release 2022.02.3). We used the Cliff's Delta analysis method (a non-parametric statistical test) to measure the effect size. An effect size of +1.0 or -1.0 indicates that there is no overlap between the distribution of precision, recall, F1-score, and MCC. An effect size of 0.0 indicates that the distribution between each pair of evaluation metrics overlaps completely.

3.6 Prediction Model

In this study, we chose to employ a random forest (RF) model for learning build outcome prediction. This was mainly because RF 1) has a white-box nature that can be utilized to extract the set of features that influences the prediction, and 2) tends to perform well with discrete and high-dimensional input data [6]. The hyperparameters of the model were kept in their default state as found in the scikit-learn library (version 0.20.4). The only alteration that we made was in the n_estimator (the number of trees) parameter, where we changed its value from 10 to 100. This was a design choice that we made based on the findings reported in a previous study [1] in which the authors experimented the use of an RF for predicting test case execution outcomes. The findings showed that using an RF model with the same default parameters would outperform four other deep learning and tree models in test case outcome prediction.

3.7 Experimental Subjects and Class Balancing

We began the experiment by applying 10-fold stratified cross validation on the build records to create the experimental subjects. Each generated subject (fold) was used for validating the RF classifier, which we trained on the remaining nine folds for each TSM metric. Similarly, 10-fold stratified cross validation was applied on the set of code changes that we extracted from each project.

One aspect that is known to affect the performance of predictive models is related to class imbalance, where the number of training instances in the data for one class outnumbers instances that belong to the other class [3]. The effect of training a model on imbalanced data-set lies in creating a model that is biased towards one of the classes. In order to control the effect of this aspect, we achieved a balanced distribution of build records and lines of code in the minority class of of each training fold in every analyzed project. To that end, we used the 'resample' module provided in the Scikit-learn library [12] whenever more than 50% of build records or lines of code at each project belonged to either one of the binary classes. Note that the resampling procedure was applied to the training data-sets only, as we wanted to evaluate the model's generalizability on real-world scenarios where data-sets come unbalanced.

4 RESULTS

This section describes the results of the statistical tests conducted to evaluate the four hypotheses and answer the research question.

4.1 Evaluation of Metrics effectiveness

To evaluate the effectiveness of the TSM and TF metrics, we begin by calculating the descriptive statistics of the precision, recall, F1-score, and MCC for the RF model that we trained on each fold in every analyzed project. Table 4 presents descriptive statistics describing the mean and standard deviation (SD) of precision, recall, F1-score, and MCC for the total number of folds (N) in the entire set of analyzed projects. The descriptive statistics reveal that learning build prediction is most effective when using the token frequency and the `gh_num_commits_on_files _touched` metrics [4]. While the token frequency metric slightly outperformed the `gh_num_ commits_on_ files_touched` metric with respect to precision, recall, and F1-score, the `gh_num_commits_on_files_touched` metric surpassed the latter with respect to MCC. A big difference between F1 and MCC can happen if the classes in the data-set are not balanced. While we balanced the training data-set, we did not for the test data-set as explained above. Hence, the number of failing lines that are falsely predicted as passing by `TF` is fairly small compared to the lines correctly predicted to pass and incorrectly predicted to fail, a line that is failing is not unlikely to be predicted as passing using `TF`. Figure 1 is a bar plot that visualizes the mean scores of the four evaluation metrics for each software metric across the 117 projects. The x-axis represents the metric names, and the y-axis corresponds to the evaluation metrics' values. From the dotted frame in Figure 1, it can be seen that by far the greatest mean precision, recall, and F1-score were achieved when using the gh_num_ commits_on_ files_touched and the token frequency metrics.

[4]We are aware of the wide spread in the distribution of precision, recall, F1-score, and MCC (high SD) values. Therefore, we used non-parametric statistical tests for comparing the distribution of values.

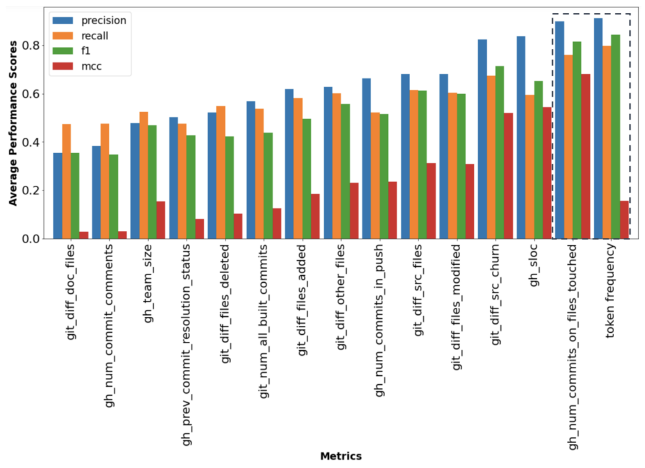


Figure 1: Mean Performance Scores of Each Metric.

To gain a better understanding of the effectiveness of each metric, we plotted the distribution of precision, recall, F1-score and MCC values for every project. Figures 2a, 2b, 2c and 2d are boxplot charts that visualize the distributions. By inspecting the distributions, we observe the following:

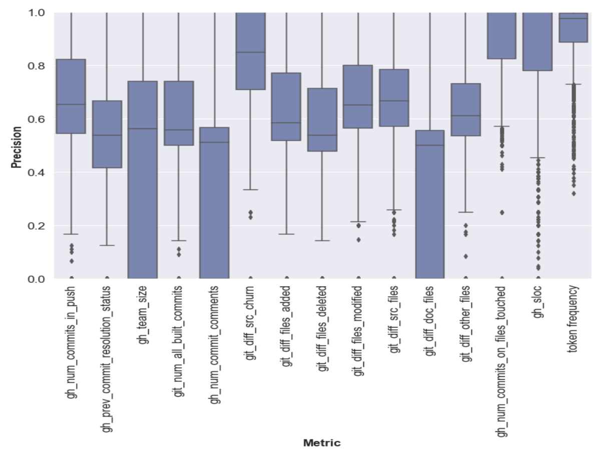
- there exists a large disparity in the distribution of the four evaluation metrics with respect to the majority of the examined software metrics.
- the `TF` metric yields better MCC scores than all the other metrics in several other projects.
- the lowest attained precision, recall, and F1-score values when using `TF` is higher than the lowest value attained when using the other TSM metrics.

4.2 Hypotheses Testing

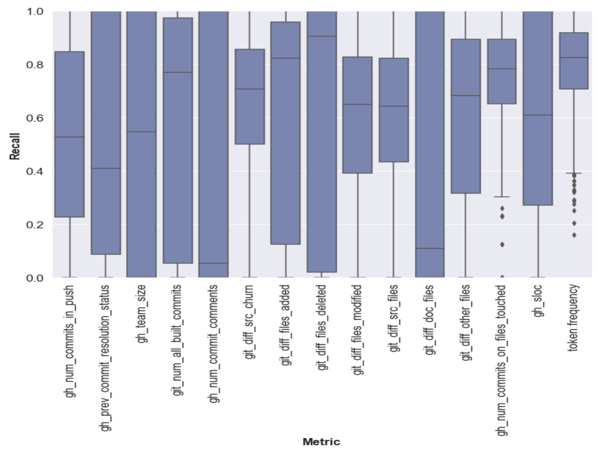
4.2.1 Significance Testing. We begin the hypotheses testing by conducting a normality test for the distribution of the four evaluation metrics. The statistical test results of normality for the four evaluation metrics when learning a classifier from the 15 software metrics show that the assumption of normality can be rejected for all the four evaluation metrics (p <0.05). Therefore, we decided to use a non-parametric statistical test for testing the hypotheses. To answer the research question of *How effective is the token frequency metric in comparison with traditional software metrics for predicting build outcomes in CI?*, we started the analysis by running the Kruskal-Wallis test for comparing the distribution of the evaluation metrics attained when using the TSM and TF metrics. The Kruskal-Wallis test results show that there is a statistically significant difference between the precision, recall, F1-score, and MCC variables with respect to the 15 software metrics (p-value <0.05). Table 5 summarizes the Kruskal-Wallis test results for each evaluation metric respectively. Since the Kruskal-wallis test is an omnibus statistical test i.e., it cannot tell which variable is statistically significantly different, we decided to run a Dunn's post hoc statistical test. To

Table 4: Descriptive Statistics for the Precision, Recall, F1-score, and MCC

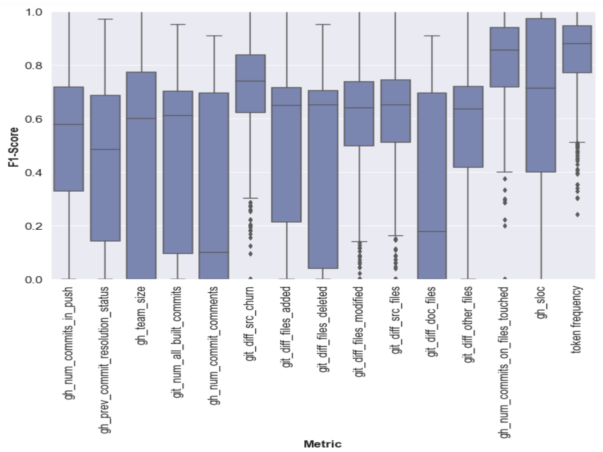
		Precision		Recall		F1-score		MCC	
Metric	**N**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**	**Mean**	**SD**
gh_num_commit_comments	1170	0.38	0.35	0.48	0.49	0.348	0.353	0.03	0.08
gh_num_commits_in_push	1170	0.66	0.25	0.52	0.33	0.516	0.257	0.24	0.25
gh_num_commits_on_files_touched	1170	0.90	0.15	0.76	0.17	0.816	0.157	0.68	0.27
gh_prev_commit_resolution_status	1170	0.50	0.30	0.48	0.39	0.427	0.288	0.08	0.18
gh_sloc	1170	0.84	0.27	0.60	0.36	0.653	0.325	0.55	0.41
gh_team_size	1170	0.48	0.35	0.52	0.43	0.470	0.357	0.15	0.40
git_diff_doc_files	1170	0.36	0.33	0.47	0.48	0.354	0.350	0.03	0.10
git_diff_files_added	1170	0.62	0.27	0.58	0.41	0.495	0.287	0.19	0.20
git_diff_files_deleted	1170	0.52	0.33	0.55	0.47	0.422	0.334	0.10	0.16
git_diff_files_modified	1170	0.68	0.21	0.60	0.27	0.601	0.209	0.31	0.26
git_diff_other_files	1170	0.63	0.22	0.60	0.33	0.558	0.240	0.23	0.23
git_diff_src_churn	1170	0.82	0.18	0.67	0.22	0.715	0.176	0.52	0.26
git_diff_src_files	1170	0.68	0.20	0.61	0.26	0.614	0.198	0.31	0.25
git_num_all_built_commits	1170	0.57	0.30	0.54	0.44	0.440	0.311	0.13	0.18
token frequency	1170	0.91	0.13	0.80	0.15	0.846	0.133	0.16	0.21



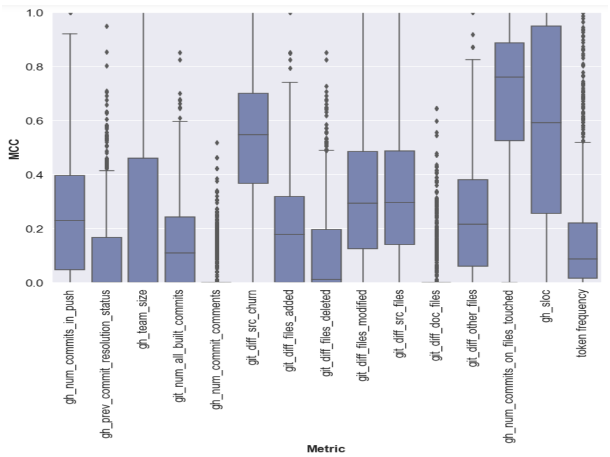
(a) Precision.



(b) Recall.

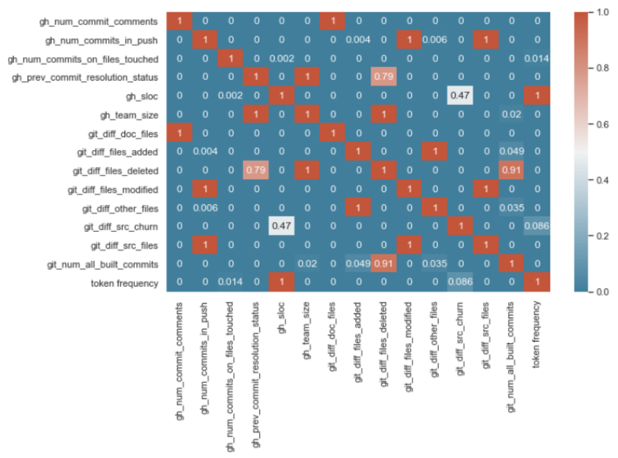


(c) F1-score.

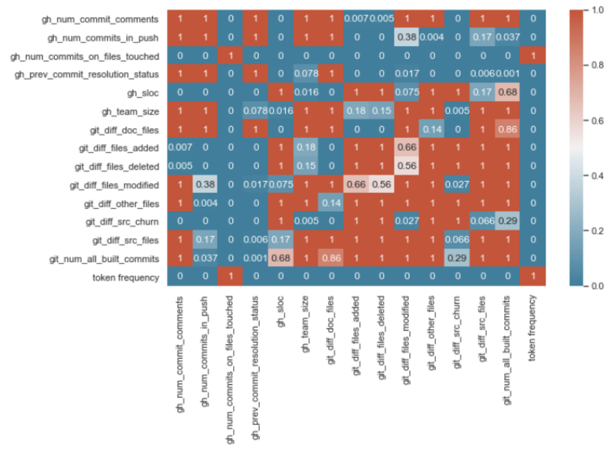


(d) MCC.

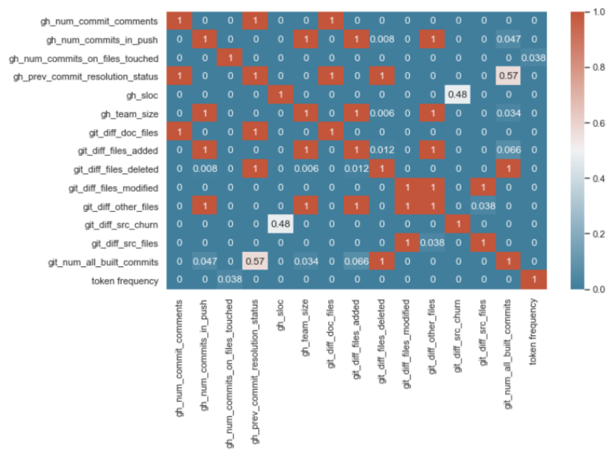
Figure 2: The Distribution of the Evaluation Metrics For the TSM and TF metrics Across All Analyzed Projects.



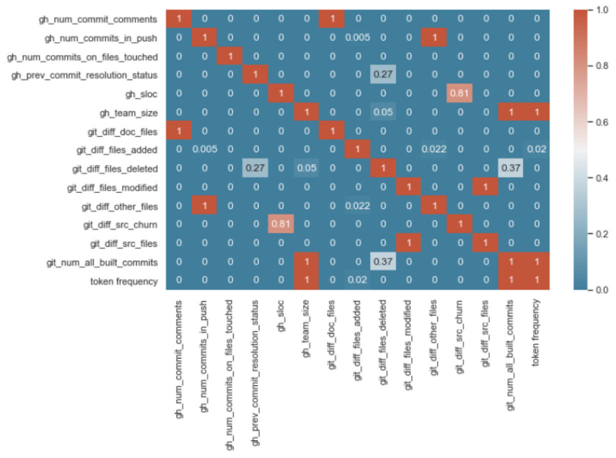
(a) Precision.



(b) Recall.



(c) F1-score.



(d) MCC.

Figure 3: Heatmaps showing the distribution of p-values when performing pairwise comparisons between the scores of each evaluation metric for each pair of metrics. Darker cells indicate smaller p-values, and lighter cells indicate larger p-values. Orange cells indicate no statistically significant difference.

control possible family-wise error rate that may occur as a result of performing multiple pairwise comparisons, we used the Bonferroni correction method for adjusting the p-values.

Table 5: The Kruskal-Wallis Test Results For Comparing the Performance Values of All Software Metrics.

Evaluation Metric	Precision	Recall	F1-score	MCC
Kruska Wallis H	5096.809	442.331	4212.293	6134.276
Sig.	<0.05	<0.05	<0.05	<0.05

Figures 3a, 3b, 3c and 3d are heatmap plots that visualize the distribution of p-values obtained from each post hoc pairwise comparison between the precision, recall, F1-score and MCC variables. The lower the p-value between each pair of software metrics (<0.05), the more confident we can be that there is a statistically significant difference between them. By inspecting the p-values in Figures 3a, 3b, 3c and 3d, we draw the following observations:

- predicting build outcomes using the token frequency or the `gh_num_commits_on_files_touched` metrics results in a statistically significantly different recall and F1 scores than those attained when using each of the other TSM metric (with p <0.05).
- the precision scores attained when using the token frequency metric is significantly different compared to the precision scores attained when using the majority of the other software metrics. The only two exceptions were with the `git_diff_src_churn` and the `gh_sloc` metrics, where no statistically significant difference could be drawn.
- using the `gh_num_commits_on_files_touched` results in a statistically significant difference with respect to MCC compared with all the other examined metrics (p <0.05).

Based on these observations, the hypothesis that *The mean precision is the same for a model trained on token frequency and each traditional software metrics (H_{0p})* can be rejected except for the `git_diff_src_churn` and the `gh_sloc` metrics, since no significant difference was captured with these. This observation brings us to believe that using the token frequency metric is more effective than twelve of the fourteen other traditional software metrics in identifying passing builds. Similarly, our results reveal that the precision scores recorded when training a model on the `gh_num_commits_on_files_touched` metric were significantly different than all the other precision scores attained when using each software metric. Thus, we observe that using the count of unique commits on the files included in builds within 3 months of last commit is a more reliable predictor for identifying passing builds, compared to the other metrics.

On the other hand, the hypotheses that *The mean recall and F1-score are the same for a model trained on token frequency and each traditional software metrics (H_{0r} and H_{0f})* can be rejected. This is because a statistically significant difference was captured between the recall scores attained when using token frequency and every other software metric. Hence, using the token frequency metric for training a classifier on predicting build outcomes is more effective for identifying the highest amount of relevant builds that will pass.

Similarly, the hypothesis that *the mean MCC is the same for a model trained on token frequency and each traditional software metrics (H_{0mcc})* can be rejected for all of the other traditional software metrics except for the `gh_team_size` and the `gh_num_all_built_commits`, which were not statistically significantly different with the MCC scores attained by the token frequency metric.

4.2.2 Effect Size. Table 6 summarizes the effect size results for the TF and `gh_num_commits_on_files_touched` metrics. The `gh_num_commits_on_files_touched` metric was chosen since it outperformed the other TSMs with respect to precision, recall, F1-score, and MCC. While the calculated p-values in Figures 3a, 3b, 3c, and 3d indicate that there is a statistically significant difference between the precision, recall, and F1-score produced when using `TF` and `gh_num_commits metrics`, the Cliff's Delta analysis shows that the difference in effect size between the two metrics is relatively small (<|0.3|). On the other hand, the effect size between the MCC scores was found to be large, indicating a large difference when using the two metrics respectively (>|0.8|).

Table 6: The Cliff's Delta Analysis Results Between the Four Evaluation Metrics, Comparing Token Frequency and gh_num_commits_on_files_touched

Name	Effect size (delta_estimate)	Lower	Upper
Precision	-0.23	-0.28	-0.18
Recall	0.12	0.07	0.17
F1	0.09	0.04	0.14
MCC	-0.83	-0.85	-0.80

While the difference in effect size between `TF` and `gh_num_comments_on_files_touched` is small with respect to precision, recall, and F1-score, the advantage that `TF` has over the TSM is the fact that it operates on a fine-grained level, which allows developers to pinpoint lines of code that require debugging before committing new code changes. A line of code example from the 'Cloudify' project [5] that was correctly identified by the TF-trained model to trigger a build failure is `"\t\t\t\t\tif (!(Boolean) session.get(Constants.INTERACTIVE_MODE)) {"`.

RQ. How effective is the token frequency metric in comparison with traditional software metrics for predicting build outcomes in CI?

Our experiment on 117 Java projects revealed that using the token frequency metric for training a classifier on build outcome prediction yields a slightly better predictive quality for the passed builds than when using the best traditional software metrics. On the other hand, the majority of the examined traditional software metrics were found to yield higher predictive quality than the token frequency metric when it comes to the failed builds.

Overall, the findings of this study suggest that both line and file level metrics are effective for learning build outcome predictions in CI. However, the usage of each metric type may vary depending on the business needs and target domain in which the system operates. For instance, practitioners that would prioritize fast releases over capturing issues in the system might opt to use line-level metric for training as it allows developers to be more confident to start using the code base for implementing new features without waiting for the build to finish running if it was predicted to pass. This assumption needs to be validated in future studies. On the other hand, if practitioners are working on developing safety critical systems, then capturing all issues in the system is prioritized over fast releases. In this case, using the file level metrics is more desirable since those were shown to yield higher effectiveness rate in alerting developers about issues in the code that require fixing.

5 THREATS TO VALIDITY

In this section, we discuss the limitations of our paper using the framework recommended by Wohlin et al. [15].

External Validity: Our study investigates the effectiveness of fifteen different software metrics in predicting build outcomes for 117 Java projects. Hence, we are aware that we can not generalize the conclusions drawn in this study to projects that are written in different programming languages. The results reported in this study may vary if we observe projects that are written in other languages or ones that are linked with different CI services. New studies are needed to validate the effectiveness of the TF metric for projects written in different languages and CI services.

Construct validity: The most tangible threat to construct validity is that we can not assert whether all collected build records with 'failed' status were due to, for example, environmental breakages or faults in the code. Hence, the likelihood of analyzing build records that failed due to factors that are not related to the code-base can not be ruled out. Another threat lies in the validity of the number of build records that we extracted from the TravisTorrent data-set, and whether these records truly mirror the actual builds in Github.

[5] https://github.com/CloudifySource/cloudify

However, we minimize these threats by collecting and analyzing a large sample of build records and lines of code from 117 projects.

Internal Validity: A potential internal threat is the presence of undetected flaws in the measurement tools used for extracting both the TSM and the TF. This threat was reduced by carefully inspecting the scripts and testing them on different subsets of data.

Conclusion validity: The main conclusion drawn from this empirical study suggests that using the a line-level metric produces similar predictive quality as file-level metrics in predicting passing builds. This conclusion is based on measuring the effect size of TF and one file-level metric on four predictive performance metrics. Hence, the results might differ if we measure the effect size of TF and another file-level metric, such as gh_sloc. However, we chose the gh_num_comments_on_files_touched metric since it scored highest in mean precision, recall, F1-score, and MCC values, and since it showed a significant difference with almost every metric.

6 CONCLUSION AND FUTURE WORK

In this paper, we conducted an empirical study to examine the effectiveness of a line-level metric in learning build outcome prediction, and compared its effectiveness with fourteen traditional software metrics. We found a a small difference in effect size between token frequency and the best traditional software metrics metric for the passed builds, and a large difference for the failed builds. We conclude that using the line-level metric for training a classifier on build outcome prediction is slightly more effective than the file level metrics for the passed builds, but substantially less effective when it comes to the failed builds. The benefit that token frequency has over traditional software metrics is its ability to pinpoint lines of code that require fixing.

While our analysis revealed promising results regarding the effectiveness of line-level metric for build prediction, future work aims at analyzing additional software metrics such as those related to testing (e.g., gh_tests_ added) and others that operate on line-level. Another future direction is to experiment the use of token frequency on software written in different programming languages. Finally, we would like to evaluate the effectiveness of using both types of metrics - line and file levels - on the predictive quality of a model for build outcome predictions in CI.

REFERENCES

[1] Khaled Al Sabbagh, Miroslaw Staron, Regina Hebig, and Wilhelm Meding. 2019. Predicting Test Case Verdicts Using TextualAnalysis of Commited Code Churns. In *CEUR Workshop Proceedings*, Vol. 2476. 138–153.

[2] Khaled Walid Al-Sabbagh, Miroslaw Staron, Regina Hebig, and Wilhelm Meding. 2020. Improving Data Quality for Regression Test Selection by Reducing Annotation Noise. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 191–194.

[3] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. 2004. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter* 6, 1 (2004), 20–29.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2016. *Oops, my tests broke the build: An analysis of travis ci builds with github.* Technical Report. PeerJ Preprints.

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 447–450.

[6] Matthias Feurer and Frank Hutter. 2019. Hyperparameter optimization. In *Automated machine learning*. Springer, Cham, 3–33.

[7] Ahmed E Hassan and Ken Zhang. 2006. Using decision trees to predict the certification result of a build. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 189–198.

[8] Foyzul Hassan and Xiaoyin Wang. 2017. Change-aware build prediction model for stall avoidance in continuous integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 157–162.

[9] Yang Luo, Yangyang Zhao, Wanwangying Ma, and Lin Chen. 2017. What are the factors impacting build breakage?. In *2017 14th Web Information Systems and Applications Conference (WISA)*. IEEE, 139–142.

[10] Ansong Ni and Ming Li. 2017. Cost-effective build outcome prediction using cascaded classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 455–458.

[11] Miroslaw Ochodek, Miroslaw Staron, Dominik Bargowski, Wilhelm Meding, and Regina Hebig. 2017. Using machine learning to design a flexible LOC counter. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 14–20.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[13] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 345–355.

[14] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*. 724–734.

[15] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

[16] Jing Xia and Yanhui Li. 2017. Could we predict the result of a continuous integration build? An empirical study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 311–315.

[17] Jing Xia, Yanhui Li, and Chuanqi Wang. 2017. An Empirical Study on the Cross-Project Predictability of Continuous Integration Outcomes. In *2017 14th Web Information Systems and Applications Conference (WISA)*. IEEE, 234–239.

[18] Jingxiu Yao and Martin Shepperd. 2020. Assessing software defection prediction performance: Why using the Matthews correlation coefficient matters. In *Proceedings of the Evaluation and Assessment in Software Engineering*. 120–129.

[19] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhénuc. 2017. Do Not Trust Build Results at Face Value-An Empirical Study of 30 Million CPAN Builds. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 312–322.