A MACHINE LEARNING APPROACH FOR FAILURE PREDICTION IN CONTINUOUS INTEGRATION

ÖMER ÖZDEMIR

ÖZYEĞİN UNIVERSITY

JULY, 2025

A MACHINE LEARNING APPROACH FOR FAILURE PREDICTION IN CONTINUOUS INTEGRATION

by

Ömer Özdemir

Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in
Computer Science

Advisor: Prof. Hasan Sözer
Co-Advisor: Assoc. Prof. Reyhan Aydoğan

Graduate School of Science and Engineering
Özyeğin University
İstanbul

July, 2025

A MACHINE LEARNING APPROACH FOR FAILURE PREDICTION IN CONTINUOUS INTEGRATION

Approved by:

Prof. Hasan Sözer, Advisor
Department of Computer Science
Özyeğin University

Assoc. Prof. Ayşe Tosun Kühn
Department of Computer Engineering
Istanbul Technical University

Assoc. Prof. Reyhan Aydoğan,
Coadvisor
Department of Artificial Intelligence and Data Engineering
Özyeğin University

Prof. Geylani Kardaş
International Computer Institute
Ege University

Assist. Prof. İlknur Karadeniz
Department of Artificial Intelligence and Data Engineering
Özyeğin University

Approval Date: May 13, 2025

DECLARATION OF ORIGINALITY

I hereby declare that I am the sole author of this thesis and that this is the true copy of my thesis, including the final revisions, approved by my thesis committee. All data and information have been obtained, produced, and presented in accordance with the rules of research ethics and principles of academic honesty. As required by these rules, to the best of my knowledge I have acknowledged ideas, thoughts, and any copyrighted material in accordance with the standard referencing rules. I certify that any part of this thesis has not been submitted for a degree or diploma in another educational institution.

Ömer Özdemir

ABSTRACT

Continuous Integration (CI) is a development practice where developers regularly merge their code changes into a central repository, enabling simultaneous collaboration across a shared codebase. This frequent integration and automated building process in CI helps to detect and resolve conflicts or errors early in development. However, in large-scale systems, the build process can be costly. Each build incurs expenses, while skipping builds can increase the risk of undetected failures. This paper presents an empirical study within an industrial setting, investigating the use of machine learning techniques to predict build failures. Accurate predictions can help to identify builds that can be safely skipped to reduce CI costs. We evaluate various models and feature combinations on a dataset derived from real-world industrial projects. We observe high precision but low recall in predicting failed builds, allowing hundreds of successful builds to be correctly skipped, with around a dozen failures potentially being missed.

Keywords: Build failure prediction, Collective changes, Machine learning, CI/CD, Industrial case study

ÖZET

Sürekli Entegrasyon (CI), geliştiricilerin kod değişikliklerini düzenli olarak merkezi bir depoya birleştirdiği ve paylaşılan bir kod tabanı üzerinde eşzamanlı iş birliğini mümkün kılan bir geliştirme uygulamasıdır. CI içerisindeki bu sık entegrasyon ve otomatik derleme süreci, geliştirme aşamasının erken safhalarında çakışma veya hataların tespit edilmesine ve çözülmesine yardımcı olmaktadır. Ancak, büyük ölçekli sistemlerde derleme süreci maliyetli olabilmektedir. Her bir derleme belirli bir maliyet yaratırken, derlemelerin atlanması tespit edilemeyen hataların riskini artırabilir. Bu makale, endüstriyel bir ortamda gerçekleştirilen ampirik bir çalışmayı sunmakta ve derleme hatalarını tahmin etmek amacıyla makine öğrenmesi tekniklerinin kullanımını incelemektedir. Doğru tahminler, CI maliyetlerini azaltmak için güvenle atlanabilecek derlemelerin belirlenmesine yardımcı olmaktadır. Gerçek dünyadaki endüstriyel projelerinden türetilen bir veri kümesinde çeşitli modeller ve özellik kombinasyonları değerlendirilmektedir. Başarısız yapıları tahmin etmede yüksek hassasiyet ancak düşük duyarlılık gözlemlenmiştir, bu da bir miktar hatanın potansiyel olarak gözden kaçmasına rağmen yüzlerce başarılı yapının doğru şekilde atlanmasına olanak sağlamaktadır.

Anahtar Kelimeler: Derleme hata tahmini, Toplu değişiklikler, Makine öğrenmesi, CI/CD, Endüstriyel vaka çalışması

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisors, Assoc. Prof. Reyhan Aydoğan and Prof. Hasan Sözer, for his invaluable guidance and support throughout this journey. I am also grateful to Vestel Electronics for sharing their dataset with me, as well as for supporting my experiments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

AUC Area Under the Curve

CD Continuous Delivery / Continuous Deployment

CDF Categorized Developer Features

CI Continuous Integration

CV Cross-Validation

DCF Different Count Features

DT Decision Tree

FN False Negative

FP False Positive

FTF File Type Features

GBM Gradient Boosting Machine

I Industrial

KNN K-Nearest Neighbors

LOC Lines of Code

LR Logistic Regression

LSTM Long Short-Term Memory

ML Machine Learning

MLP Multi-Layer Perceptron

NB Naive Bayes

NN Neural Network

OS Open Source

PCA Principal Component Analysis

PR-AUC Precision-Recall Area Under Curve

PSF Previous Status Feature

RF Random Forest

RQ Research Question

SCF Source Code Features

SVM Support Vector Machine

SVN Subversion

TF/IDF Term Frequency–Inverse Document Frequency

TP True Positive

TV Television

1. INTRODUCTION

Industrial software development companies often prefer Continuous Integration (CI) systems to speed up delivery times [2, 3, 4]. These systems automate the build and test processes as code changes made by users are integrated into a common repository. This automation makes it easier to detect integration issues at an early stage, reducing the risk of errors in the final product. In particular, nightly builds are widely used, allowing extensive automated testing to be run throughout the night. These builds ensure that the codebase remains stable, functional, and ready for deployment [5, 6]. CI is especially useful in collaborative environments with multiple contributors, where integration scenarios can be complex and prone to errors [7, 8].

CI systems require frequent building of projects, which is especially challenging for building multi-layered systems, such as Android-based platforms contributed by different stakeholders. It may take extensive time to build the entire platform. Prolonged build durations can significantly extend the CI testing process, limiting the number of projects that can be evaluated even with nightly builds [9]. Therefore, foreseeing build failures in advance and skipping builds predicted to be successful may increase the efficiency of CI systems [10, 11].

In this thesis, we investigate the effective use of Machine Learning (ML) models to predict build failures in an industrial setting accurately. While many studies have focused on open source projects, only a recent few have investigated this topic in industrial settings [12, 13]. Our study is based on a case study of Android-based digital TV software projects within the consumer electronics domain. We perform extensive empirical analyses using various models trained with combinations of existing and novel features. These features include those that are derived from the categorization of developers and modified files. In contrast, existing studies that rely solely on developer identities and modified files (rather than their feature-based representations) are not applicable in dynamic industrial

environments, where both developers and files change frequently over time [14, 15].

Another important difference of our study is that it focuses on predicting build outcomes involving multiple changes by developers, rather than build outcomes triggered by a single change [16, 17, 18, 19]. In large-scale, multi-layered industrial projects, where software builds are time-consuming, taking action for every individual code change is often impractical [20]. As a result, CI tests that encompass contributions from multiple developers and their collective changes become unavoidable. Our evaluation also considers the temporal structure of real-world CI processes. Supervised machine learning models are typically evaluated by splitting data into training and testing sets. Although random data splitting yielded promising results in prior studies [21, 22, 23], we advocate that chronological data splitting offers a more realistic and reliable assessment. We observe high precision but low recall in predicting failed builds in general, resulting in F1 scores of up to 60%. This trade-off allows hundreds of successful builds to be correctly skipped, with around a dozen failures potentially being missed. The contributions of this study are summarized as follows:

- An ML-based approach to predict build failures in CI that comprise collective changes contributed by multiple developers.
- Categorization of developers and modified files to obtain features that are resilient to dynamic changes in industrial environments.
- Release of a new, anonymized dataset derived from industrial software projects, made publicly available.
- A comprehensive evaluation with various ML algorithms and feature combinations.

The remainder of this paper is organized as follows. We summarize the related work in Section 2. We explain the overall process and the dataset in Section 3. We

detail our design choices, experimental setup and challenges in Section 4. We present and discuss the results in Section 5. Finally, we conclude the paper in Section 6.

2. RELATED WORK

Most studies on build failure prediction primarily focus on utilizing features related to source code changes such as number of modified lines and number of added code lines [12, 13, 24, 25, 26, 21, 22, 23, 27]. In addition to these features, other studies take into account number of modified files or directories [13, 25, 26, 22], developer profiles [26, 21, 22], and the previous build status [12, 13, 24] in order to increase prediction accuracy. However, previous studies have not analyzed the effect of combinations of these features on build failure prediction by testing them with various ML algorithms in an industrial setting. This study aims at filling this gap. There exits studies that highlight the importance of features related to either developers or file modifications in predicting build failures [28, 29, 30]. For instance, some studies [26, 21, 22] consider the seniority and experience level of developers as predictive features. However, there is no study to our knowledge that categorizes both developers and the modified files to assess their impact on prediction accuracy in dynamic industrial settings. To fill this gap, we categorized various data related to developers and modified files. Specifically, we classified modified files based on their types (i.e., file extensions), as this aspect has been largely overlooked in the current literature on build failure prediction. We empirically study the effect of these categorizations on prediction accuracy.

Table 2.1 presents and overview and comparison of existing studies for build failure prediction. We observe that generally a small number of ML algorithms or deep learning models were used [31]. In our study, we test a wide range of both traditional and ensemble (boosting) algorithms. Features related to source code changes are utilized as features by all studies. However, a different subset of the remaining features is utilized by each study.

Existing approaches proposed for build failure prediction react on individual code changes, but a high volume of changes can lead to increased costs. To reduce these high

Table 2.1: *Comparison of CI Build Failure Studies Adapted from [1].*

Criteria	Our work	Kawalerowicz et al. [12]	Hong et al. [13]	Saidani et al. [25]	Pan et al. [26]	Yaraghi et al. [21]	Abdalkareem et al. [22]	Lee et al. [23]
Data Source	Vestel Projects (I)	Hidden Projects (I)	Atlassian Projects (I)	Travis Torrent (OS)	Apache (OS)	GitHub (OS)	GitHub (OS)	Samsung Projects (I)
Source Code Change Feature(s)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Number of modified file(s) or directory	**Yes**	No	**Yes**	**Yes**	**Yes**	No	**Yes**	No
Commit Messages	No	No	No	No	No	**Yes**	**Yes**	No
Previous Build Status	**Yes**	**Yes**	**Yes**	No	No	No	No	No
Categorized Developer	**Yes**	No	No	No	**Yes**	**Yes**	**Yes**	No
Type of the modified files	**Yes**	No	No	No	No	No	No	No
Feature Engineering	Multilabel Binarizer, Frequency encoding	Not reported	Not reported	Not reported	Not reported	Scaling and Weighting - Normalization and TF/IDF	Weighting -TF/IDF	Tagging -word2vec
Learning algorithm(s)	LR, KNN, DT, SVM, RF, GBM, XGBoost, LightGBM, AdaBoost	RF	Logistic Regression	LSTM - RNN	DT, RF, MLP, SVM, NB, LR	RF, SVM, XGBoost	DT (C4.5)	NN
Evaluation methods / Metrics	Expanding Window CV / F1, Precision, Recall, Precision-Recall AUC	Accuracy, F1, MCC and AUC	Accuracy, Precision, Recall, F1	Expanding Window CV / Accuracy and AUC	Stratified K-fold, Windows Cross and Time Series CV / FP, FN, and TP	Random K-fold / APFDC	Random K-fold / F1, Precision, Recall and AUC	Random K-fold / F1, Precision, Recall and Accuracy

costs, Jin et al. proposed two new approaches called SmartBuildSkip [32] and PreciseBuildSkip [33]. Both approaches are based on ML algorithms, with PreciseBuildSkip additionally leveraging rule-based methods. They have been evaluated on large-scale open source datasets and have provided significant cost reductions by predicting failed builds and skipping the others. Kamath et al. [34] combine two heuristics: grouping multiple builds and skipping those predicted to succeed. However, their approach still predicts build failures at the level of individual commits. In contrast, our study focuses on collective changes made by multiple developers. Some recent studies [12, 25] have addressed the case of multiple changes made by a single developer. However, these studies do not explicitly address collaborative, multi-developer contributions within a build.

There exist recent studies that investigated various factors affecting build failure prediction. Kola et al. [35] examined 900,000 builds of 1,689 open source projects. They investigated the effects of code ownership on build results and observed that build success rate increases as code ownership increases. Shimmi et al. [36] investigated the effects of different code types on build failure prediction. Their prediction model showed that maintenance-related changes, especially those containing bug fixes, cause the most build failures. In addition, Hong et al. [13] conducted a large-scale industrial study at Atlassian, demonstrating that previous build results are effective predictors of build failures. Consistent with their findings, our study identifies the previous build status as the most influential feature. However, we extend our scope by providing a comprehensive analysis of multiple features and examining how their combinations affect build failure prediction across various ML algorithms.

3. PROPOSED APPROACH: OVERALL PROCESS AND THE DATASET

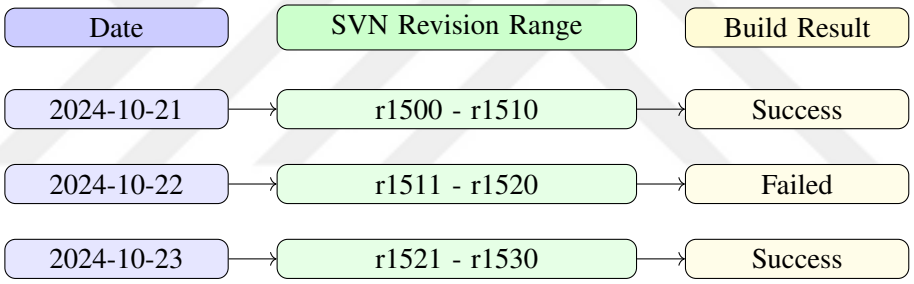
Our primary objective in this study is to investigate the effectiveness of ML in predicting the outcome of builds (i.e., whether a given build will succeed or fail) based on features derived from collective code changes across multiple software projects. Additionally, we aim to identify which ML algorithms and feature combinations yield the most accurate predictions. K-fold cross-validation has been widely employed in prior studies [21, 22, 23], wherein datasets are randomly partitioned into k subsets to evaluate model performance across varying training and testing configurations. However, for practical applications in an industrial setting, models are typically trained only on data available up to a given point in time to predict future outcomes. Therefore, we also want to test the model in a more realistic setup, where the training and testing sets are formed without breaking the actual time order of data samples. Finally, given the limited number of ML algorithms tested so far, we aim to compare a broader range, particularly to assess whether significant differences exist between traditional and ensemble (boosting) algorithms. To address these goals, we formulated the following research questions:

- **RQ1**: To what extent does the choice between random and chronological train-test splits affect the effectiveness of ML models in predicting build failures from features of collective changes?
- **RQ2**: Which ML algorithms and combinations of features lead to the most accurate predictions?
- **RQ3**: Can we improve the prediction accuracy by adopting ensemble learning approaches?

To address these questions, we conducted an industrial case study in the context

of Vestel[1], which is one of the largest TV manufacturers in Europe. Vestel produces TV products for 157 different brands from 145 different countries [37]. Therefore, there are multiple projects being maintained in different repositories. When developers commit their changes (e.g., add, modify or remove files) to the version control system (Subversion - SVN), these changes are subsequently merged daily into the main source code repository (Git). Once all project commits have been consolidated, they are processed through an Android nightly build system. The outcomes of this build process (success or failure) are recorded as illustrated in Figure 3.1. Some of the data has been used in training to build the prediction model, while the rest has been used as test data to assess the prediction accuracy of the learned model.

Figure 3.1: *A Snippet from the Nightly Build Results.*

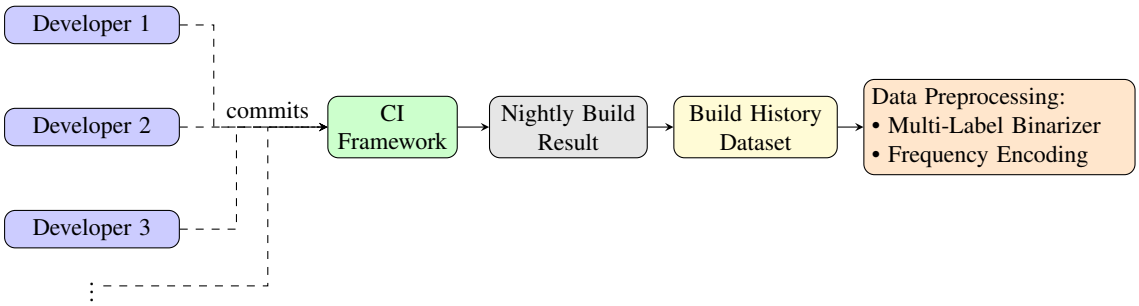


While numerical features such as the number of modified files and total addition lines are not preprocessed, discrete features such as the file type feature is transformed into binary values to indicate the presence of different file types. The feature groups representing developer characteristics, such as *Seniority Group* and *Age Group*, were expanded into sub-features based on all possible combinations they contain. In this way, the attributes of the developers involved in each build and the number of developers possessing each attribute were numerically encoded into the dataset. In summary, categorical file types were encoded using the multi-label binarizer, and developer-related categorical

[1] `http://www.vestel.com.tr`

features were encoded using frequency encoding [38, 39, 40], as illustrated Figure 3.2.

Figure 3.2: *Workflow for Generating Preprocessed Build Datasets.*



Overall, the collected features are categorized into *Source Code Features (SCF)*, *Difference Count Features (DCF)*, *Categorized Developer Features (CDF)*, *File Type Features (FTF)*, and *Previous Status Feature (PSF)* as listed in Table 3.1.

SCF represent features that contain changes made to the source code, including lines added, deleted or changed. Special emphasis is given to Android-specific code (e.g., Java or Kotlin), and changes related to the core functionality of the system are isolated [41]. In addition, SCF include Android System Version, which reflects the version of the middleware provided by the chip vendor.

DCF comprises features that numerically express daily changes of developers. It includes the number of changed files, directories and subsystems. Apart from these, since we are examining industrial Android projects, our data also includes the number of changed apps.

CDF categorizes developers according to various attributes; these include age, gender, seniority, experience level and *Checking Control Status*, which refers to information from developers about whether their changes were reviewed before being committed. Developers are categorized according to the following characteristics:

- Age Group: {young, middle-aged, mature-aged} (-30, 30-40, 40+ years)

Table 3.1: *Feature Categories and the Corresponding Data Types.*

Category Name / Corresponding Name in Literature	**Feature Name (Data Type)**
Source Code Core Features (SCF) / Size	Total Addition Lines (int) Total Deletion Lines (int) Android Code Addition Lines (int) Android Code Deletion Lines (int) Total Lines Changed (int) Android Code Lines Changed (int) Total Lines (int) Android Code Lines (int) Android System Version (int)
Difference Count Features (DCF) / Diffusion	Number of Modified Files (int) Number of Modified Directories (int) Number of Modified Subsystems (int) Number of Modified Apps (int)
Categorized Developer Features (CDF) / Experience	Age Group (string array) Gender Group (string array) Seniority Group (string array) Experience Group (string array) Checking Control Status (string array)
File Type Feature (FTF) / Purpose	Changed Files (string array)
Previous Status Feature (PSF) / History	Previous Build Status (bool)

- Gender Group: {male, female}
- Seniority Group: {pm, technologist, software architect, senior expert, advanced expert, expert, normal}
- Experience Group: {beginner, intermediate, expert} (3, 3-6, 6+ years)
- Checking Control Status: {no, mixed, yes}

CDF categorizes developers using multi-label attributes. FTF captures the types of files modified by developers. Files are categorized based on predefined patterns, such as bin files, C files, json files, makefiles, xml files, gradle files, android source code (Java or Kotlin) files, and software configuration files (e.g., with extensions like .kl, .rc, .te or .ini). Finally, the dataset includes PSF *(Previous Build Status)* that reflects the outcome (i.e., success, fail) of the previous build.

Our dataset contains over 1,500 builds from 9 different projects as seen in Table 3.2. In total, 13 developers contributed with 3,215 commits. Properties of data collected for various groups of projects is listed in Table 3.3, where projects are enumerated from P1 to P9. Grouping is based on the organization of the company. P1 exhibits the highest volume of code changes. In fact, all the projects other than P1 existed before the CI system was established, resulting in fewer code changes despite being older than P1. The total number of builds, the number of successful builds and the number of those associated with failures are listed in Table 3.4. We can see that the dataset is unbalanced, with 95.13% of the builds being successful and only 4.87% resulting in failure. Note that we have 47 features in total.

Table 3.2: *An Overview of the Collected Dataset.*

Metric	**Value**
Total Software Build Count	1,522
Total Project Count	9
Total Developer Count	13
Total Commit Count	3,215
Total Number of Files Changed	28,563
Total Lines of Code Changed (LOC)	392,648
Total Build Success	1,448
Total Build Fail	74

Table 3.3: *Collected Data for Various Project Groups.*

Project	**Total Commits**	**# of Files Changed**	**LOC Changed**
P1	695	9,140	368,856
P2-P3	1,000	8,386	2,743
P4-P7	1,474	10,216	11,342
P8-P9	46	821	9,707

The dataset as well as the source code used in this work are accessible through

Table 3.4: *The List of Projects and their Build Results.*

Project	Total Builds	Successful Builds (%)	Failed Builds (%)
P1	105	88 (83.81)	17 (16.19)
P2	348	334 (95.98)	14 (4.02)
P3	59	57 (96.61)	2 (3.39)
P4	317	300 (94.64)	17 (5.36)
P5	258	245 (94.96)	13 (5.04)
P6	315	311 (98.73)	4 (1.27)
P7	80	76 (95.00)	4 (5.00)
P8	27	25 (92.59)	2 (7.41)
P9	13	12 (92.31)	1 (7.69)
Total	1,522	1,448 (95.13)	74 (4.87)

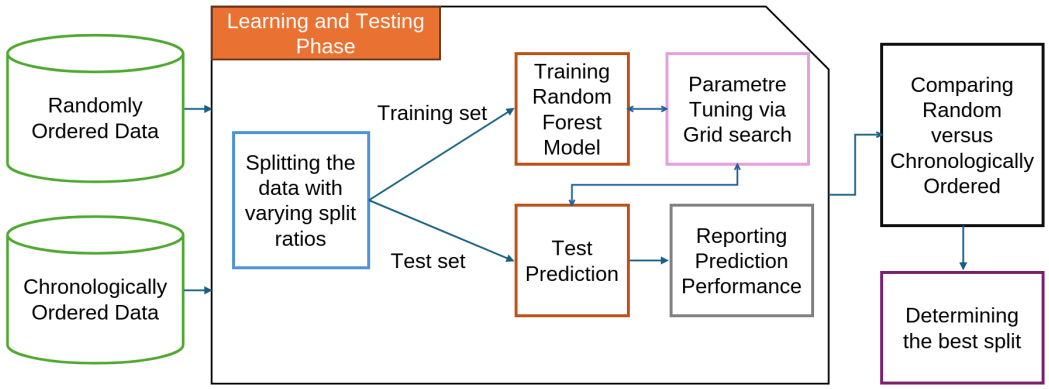
a public repository[2]. In the following section, we explain our experimental setup for evaluation.

[2] http://doi.org/10.6084/m9.figshare.28930103

4. EVALUATION, DESIGN CHOICES AND CHALLENGES

Since we advocate adopting a chronological train- test splits (i.e., trained only on data available up to a given point in time to predict future outcomes) for a more realistic setting, we first examine how data splitting (i.e., randomly shuffled or chronologically ordered data) affects the prediction performance of the models for build failure prediction. Accordingly we set up our first experiment as illustrated in Figure 4.1. The dataset is split by considering two splitting approaches to construct training and testing datasets: *i)* Randomized data with varying split ratios, and *ii)* Chronological train-test split with an expanding window. A chronological train-test split mirrors real-world scenarios by ensuring that training data precedes test data temporally, allowing predictions to be based on the relationship between historical changes and future failures [42, 36]. In this phase, we train a Random Forest model by using the training dataset. This model is tested on the test set by considering the performance metrics such as precision, recall, F1 Score, and Area Under the Precision-Recall Curve (PR-AUC).

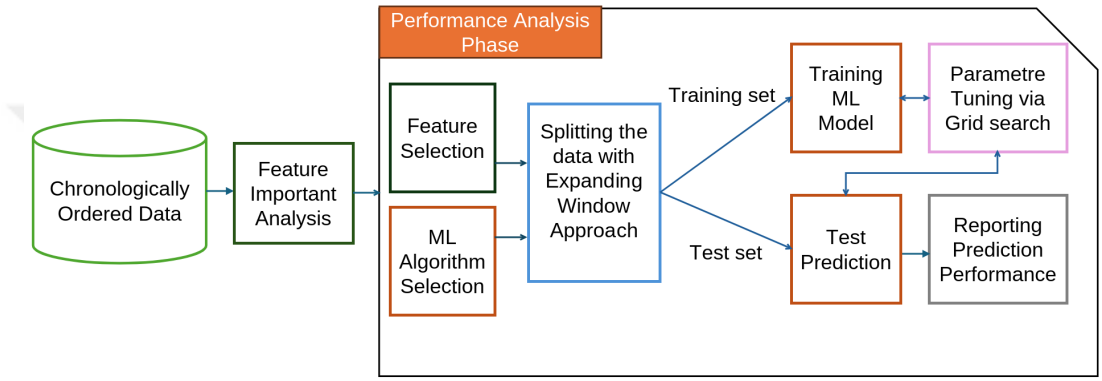
Figure 4.1: *Experimental Setup Phase-1.*



In the second, *main* experimental setup, we focus on assessing the prediction performance of different machine learning methods and ensembling techniques. As depicted in Figure 4.2, we first perform a feature importance analysis, and then for the selected

combination of features, we train a separate model on the same training dataset. The same performance metrics listed in the first phase are used to assess the performance of each model for each feature combination on the test set. It is worth noting that we adopt chronologically ordered data along with the expanding window approach for simulating a realistic temporal evaluation setting.

Figure 4.2: *Experimental Setup Phase-2.*



In order to understand the relevance of each feature in our model, we employed various feature selection and importance evaluation techniques. These techniques help identify the most influential features, reducing the dimensionality of the dataset and improving model interpretability and performance [43]. In the feature selection process, we employed the Random Forest classifier. In this thesis, we utilize three feature analysis techniques to evaluate and select the most relevant features for prediction: *i) Feature Importance* identifying key predictors based on model-derived scores; *ii) Correlation Analysis* assessing the relationships between features and the target variable; and *iii) Select K-Best*. We detail the use of these techniques as follows:

- **Feature Importance:** This method determines the importance score according to how much the feature improves performance in different decision trees [44] based on the given Random Forest model. Note that we encode FTF and CDF, which are

represented as multilabel data (e.g., arrays of strings), into numeric matrices. When calculating feature importance, the contributions of all related numeric columns were summed to obtain a single importance score for the entire multi-label feature [45].

- **Correlation Analysis:** We used Pearson Correlation Analysis to assess the linear relationship between each numerical feature and the target variable [46]. A correlation value close to 1 or -1 indicates a strong relationship, whereas a value close to 0 indicates a weak or no relationship. While calculating the correlation values, we transformed multi-label representation for FTF and CDF into a format suitable for numeric correlation analysis (using Label Encoder) [45]. This decision was motivated by the observation that features with above-average correlations hold greater significance [47].

- **Select K-Best:** We applied the Select K-Best method, a statistical method that ranks and selects the top-performing features, with the ANOVA F-test to select the top K features and to select the most relevant features for classification [48]. We present and discuss the results in the following section.

For the prediction model, we employed both traditional ML algorithms and ensemble boosting algorithms as listed in Table 4.1. Traditional algorithms are widely used for their simplicity and interpretability. Ensemble boosting algorithms combine the predictions of multiple models to achieve higher accuracy and robustness. Logistic regression is a highly effective model for binary classification [49], particularly when there is a linear relationship between the dependent and independent variables. K-Nearest Neighbor (KNN) algorithm classifies data based on the most similar examples, and it performs well when the classes are clearly separable [50]. Support Vector Machines (SVMs) are

effective for complex classification tasks, particularly when data is not linearly separable. However, they can be sensitive to imbalanced datasets, often requiring techniques such as class weighting or resampling to improve performance [51]. Decision Trees show a non-linear approach by dividing the data according to the most meaningful feature at each node, which provides transparency in the decision-making process.

Regarding the ensemble techniques, Random Forest constructs multiple decision trees using random subsets of the data. This approach reduces variance and mitigates overfitting by averaging the predictions of uncorrelated trees. Gradient Boosting and its optimized versions, XGBoost and LightGBM, build models sequentially, each correcting the errors of its predecessor. This iterative approach makes them highly effective in dealing with complex and imbalanced datasets [52]. AdaBoost, on the other hand, enhances performance by combining multiple weak classifiers, adjusting their weights to focus more on previously misclassified instances [53].

Table 4.1: *The Set of ML algorithms used for Experimentation.*

Traditional Algorithms	**Ensemble (Boosting) Algorithms**
Logistic Regression	Random Forest
K Neighbors	Gradient Boost
Decision Tree	XGBoost
SVM	LightGBM
	AdaBoost

The parameters of each algorithm used in our model must be selected separately to give the best results under all conditions. Therefore, hyperparameter optimization was performed for each algorithm and feature set using Grid Search method [54]. The F1 score value is seen here as the basic evaluation metric that allows the selection of the most appropriate parameters [55]. It is important to scale data when using certain algorithms, such as KNN and Support Vector Machines (SVMs), as these algorithms depend

on distance calculations. To achieve better predictions, we applied Standard Scaling for KNN, and Maximum Absolute Scaling for SVM, and LR.

5. RESULTS AND DISCUSSION

In this section, we first examine the accuracy of prediction for both random and chronological splitting of training and test data. Hereby, we employ the random forest algorithm, which is the mostly used algorithm for build failure prediction with relatively better results so far. Subsequently, the prediction performance of ML algorithms is examined with commonly used feature sets. Then, the set of features are analyzed with feature analysis techniques and the performance of traditional ML algorithms is evaluated with various feature sets combined based on their importance. The effectiveness of ensemble algorithms are then investigated for varying combinations of feature sets. Threats to validity are discussed at the end of the section.

5.1 Effectiveness of Prediction for Randomly and Chronologically Ordered Data (RQ1)

On one hand, most of the studies mentioned in Table 2.1, train and evaluate their predictive models using randomized datasets, primarily employing a K-fold cross-validation approach. On the other hand, a sequence of builds are not entirely time-independent of one another. Collective code changes develop over time, which means that training and testing on randomized datasets can be misleading, as the model inadvertently leverages future data instances (i.e., nearby test instances) during the training phase. In real-world scenarios, we can only use instances that have been observed thus far and assess their predictive performance on future instances that have yet to be encountered. In this context, it would be more beneficial to take the temporal relationships within the dataset into account and to operate on a chronologically ordered dataset.

Accordingly, we aimed to investigate the prediction performance for the random and chronological splitting of training and test data. For the prediction model, we used the Random Forest algorithm, as it has been widely used and shown to deliver satisfactory

performance in previous studies [12, 26, 21]. We conducted experiments using different training and test dataset split ratios: 90-10, 80-20, 70-30, and 60-40. We compared two approaches: *i) Random:* randomly selecting the training and test samples, and *ii) Chronological:* using the initial portion of the timely-sorted data for training and the remaining portion for testing. For each split, the performance on both data splits is given in Table 5.1, where the last two columns denote the number and percentage of build failures and success instances in the corresponding test set.

Table 5.1: *Prediction performance of the Random Forest algorithm for random and chronological splitting of training and test data.*

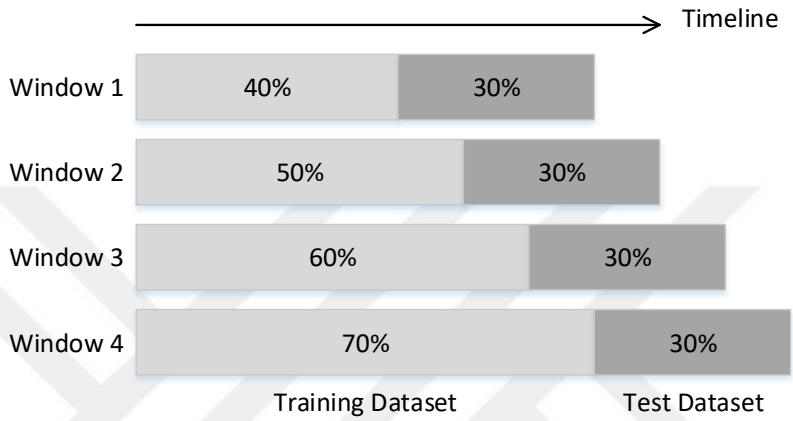
Splitting	Train-Test Ratio	Precision	Recall	F1 Score	PR-AUC	Test Success	Test Fail
Random	90-10	0.3334	0.1667	0.2223	0.2154	147 (96.08%)	6 (3.92%)
Chronological		1.0000	0.1429	0.2500	0.2190	146 (95.42%)	7 (4.58%)
Random	80-20	0.5000	0.2143	0.3000	0.2410	291 (95.41%)	14 (4.59%)
Chronological		1.0000	0.1111	0.1920	0.5250	287 (94.10%)	18 (5.90%)
Random	70-30	0.6364	0.3182	0.4242	0.2806	435 (95.19%)	22 (4.81%)
Chronological		1.0000	0.3077	0.4706	0.4306	431 (94.31%)	26 (5.69%)
Random	60-40	0.7857	0.3253	0.4583	0.3480	575 (94.42%)	34 (5.58%)
Chronological		1.0000	0.2188	0.3590	0.3505	577 (94.75%)	32 (5.25%)

Although the number/percentage of failure instances in the test sets are almost the same, the results differ depending on whether random or chronological splitting is used. Chronological splitting generally yields higher precision, while random splitting tends to produce higher recall. This result supports our claim that the performance of the prediction model is misleading due to the use of future instances in the training phase. Although random splitting has been applied in previous studies [21, 22, 23], yielding promising results, this approach is not realistic for real-world applications. Models can only be trained on data collected until a certain point of time to make predictions regarding the builds in the future.

To better reflect a real-world scenario, we tested the effectiveness of the expanding

window approach as depicted in Figure 5.1. Hereby, we initially allocated 40% of the data for the training set. Subsequently, the training set was incrementally expanded by 10% at each step. Throughout the process, the test set size was consistently maintained at 30% of the total data.

Figure 5.1: *Our Application of the Expanded Window Approach.*



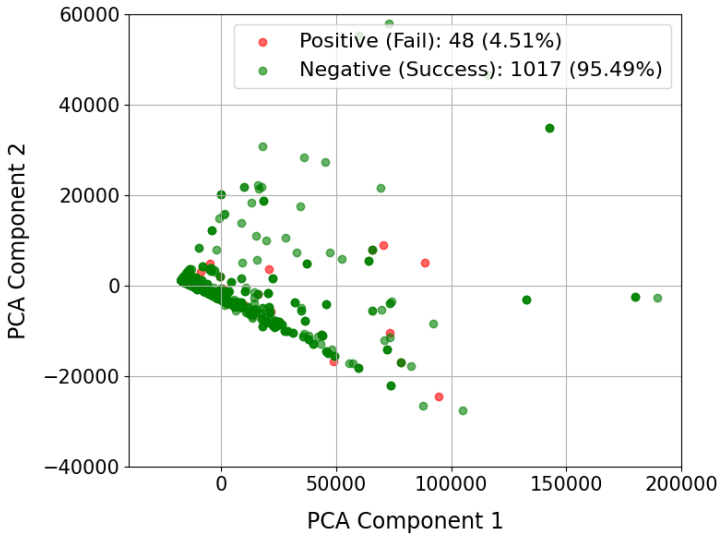
Results are listed in Table 5.2. We observe that the precision rate is significantly higher compared to recall further indicates that the learned model avoids false positives (i.e., predicting failure for a successful build), yet it often overlooks genuine positive cases (i.e., failed builds). Since the test dataset contains a minority of failing classes (i.e., positive classes), this makes it difficult for our model to predict failures. Recall that our dataset is imbalanced. That is, if we were to retrain our model to focus on detecting successful builds, we would likely see a significant increase in recall.

Table 5.2: *Results obtained with the expanding window approach.*

Window	Train Range (%)	Test Range (%)	Precision	Recall	F1 Score	PR-AUC
1	0–40	40–70	1.0000	0.1053	0.1905	0.1469
2	0–50	50–80	0.6667	0.1250	0.2105	0.2048
3	0–60	60–90	1.0000	0.2400	0.3871	0.3670
4	0–70	70–100	1.0000	0.3077	0.4706	0.4306
Avg	–	–	**0.9167**	**0.1945**	**0.3147**	**0.2873**

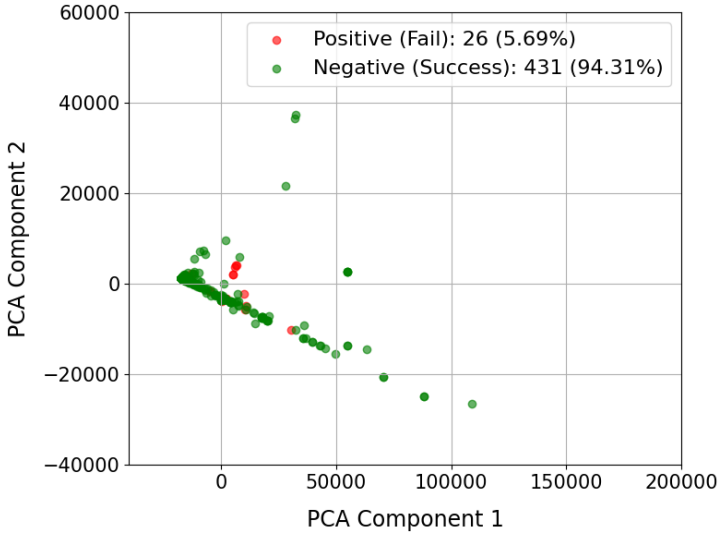
Furthermore, we investigate the results of Principal Component Analysis (PCA), where the distribution of the dataset is projected on a two dimensional plain to examine the difficulty of the prediction task for this dataset. Figure 5.2 and Figure 5.3 present the PCA visualization, illustrating the two-dimensional projection of the temporally separated training and test datasets (70-30 chronologically ordered). When the graphs are examined, it is seen that there are very few positive class examples. Also, it is seen that positive and negative class examples overlap with each other. These observations confirm that the classification task on this dataset is challenging due to both class overlap and significant class imbalance.

Figure 5.2: *Principal Component Analysis (PCA) for Training Dataset.*



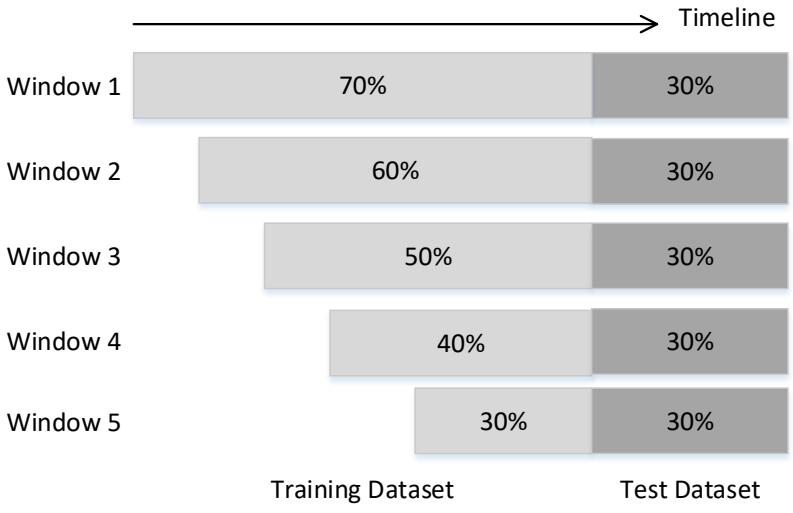
We also observe that the expanding window approach allows the model to enhance its ability to predict the minority class by progressively encountering a larger and more representative sample of minority class instances over time. Performance metrics, particularly PR-AUC scores, gradually improve as the training set size increases. One can question this interpretation since the test dataset is different for each window in our application of the expanded approach, where the window of the test dataset is shifted in

Figure 5.3: *Principal Component Analysis (PCA) for Testing Dataset.*



each iteration (See Figure 5.1). We performed another experiment to observe the effect of increasingly larger training dataset by fixing the test dataset. The approach adopted for this experiment is depicted in Figure 5.4.

Figure 5.4: *Expanding window evaluation on a fixed test set.*



The experiment depicted in Figure 5.4 evaluates the prediction performance of the model on the last part of the dataset (70-100) taken as the test dataset. The training

dataset is progressively reduced by discarding older samples, starting from the full history (0–70) down to only the most recent (40–70) portion of the dataset. Results obtained with this experiment is listed in Table 5.3, confirming that prediction accuracy improves as the training dataset grows by incorporating more historical data over time.

Table 5.3: *Results with a fixed test set as the training set is progressively reduced by discarding older samples as depicted in Figure 5.4.*

Window	Train Range (%)	Test Range (%)	Precision	Recall	F1 Score	PR-AUC
1	0–70	70–100	1.0000	0.3077	0.4706	0.4306
2	10–70	70–100	1.0000	0.3077	0.4706	0.4318
3	20–70	70–100	1.0000	0.1154	0.2069	0.3864
4	30–70	70–100	1.0000	0.1154	0.2069	0.3647
5	40–70	70–100	0.5000	0.0385	0.0714	0.3159
Avg	–	–	**0.9000**	**0.1769**	**0.2853**	**0.3859**

One might argue that the observed F1 scores (up to 0.47) are relatively low for adopting such an approach in an industrial setting. However, the approach can still be valuable depending on the specific domain, as it enables the correct prediction of hundreds of successful builds, thereby allowing them to be safely skipped, at the cost of potentially missing a dozen of failed builds. To sum up, utilizing a chronologically ordered dataset for predicting build failures based on collective code changes is more reliable. It is challenging to predict potential failures since the number of positive instances is much less than that of negative instances. Since the training and testing split ratio significantly affects the prediction performance, we suggest adopting an expanding window approach to report the generalized performance.

5.2 Effectiveness of Feature Selections and ML Algorithms (RQ2)

In the following, we report our results regarding experiments performed with traditional ML algorithms with a variety of combination of features. Then, we provide our

feature importance analysis results. Lastly, traditional ML algorithms are evaluated by progressively combining features in order of their importance, starting with only the most important feature and gradually adding less important ones.

5.2.1 Performance of ML Algorithms with Commonly Used Feature Combinations

In this section, we explore well-established traditional ML algorithms and analyze the influence of various feature sets on prediction performance. Given that it is impractical to evaluate all possible combinations of these features, we adopt an approach that involves combining some primary feature categories, particularly SCF, to examine their impact on the prediction model. SCF is regarded as a core feature category, as the features in this category are widely used in most of the ML-based CI build studies (See Table 2.1). To conduct a systematic evaluation, we first investigate the binary combinations of SCF with other categories. Ultimately, we assess the prediction performance when all available features are utilized (SCF+PSF+FTF+DCF+CDF).

Table 5.4 present the results obtained using traditional ML algorithms with various feature combinations, assessed through the expanding window evaluation technique. All evaluation metrics in the table, except for F1 Max, represent the average across window folds, while F1 Max is the highest F1 score obtained among the window folds. Notably, the highest recall value is achieved when all features are utilized in the prediction models, except for the K-Nearest Neighbor (KNN) algorithm. The best performance with KNN is observed when only the SCF and PSF features are used. Among the ML algorithms considered, KNN exhibits the poorest performance, with a recall of 0.2176. This algorithm is particularly sensitive to high dimensionality, a phenomenon known as the curse of dimensionality. In our dataset, we have a significant number of features, many of which may be irrelevant or contain noise, leading to skewed distance calculations. Additionally, KNN tends to perform poorly with imbalanced datasets, where there is a substantial disparity

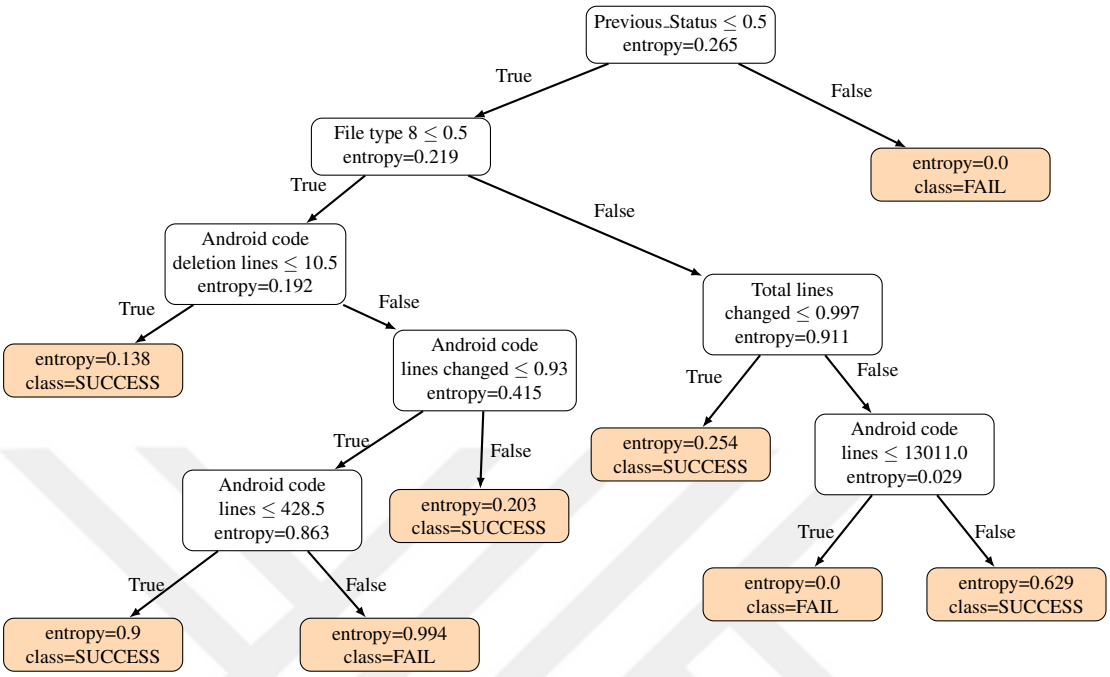
Table 5.4: *Performance Metrics for Traditional Algorithms and Feature Sets.*

Algorithm	Feature Set	Precision	Recall	F1	F1 Max	PR-AUC
Logistic Regression	SCF	0	0	0	0	0.039
	SCF + DCF	0.45	0.1057	0.1698	0.2857	0.1319
	SCF + CDF	0	0	0	0	0.1084
	SCF + FTF	0	0	0	0	0.0645
	SCF + PSF	**1**	0.2176	0.3508	0.4706	0.2756
	SCF + DCF + CDF + FTF + PSF	0.7054	**0.3586**	**0.4737**	**0.6**	**0.411**
KNN	SCF	0.1905	0.0588	0.0896	0.1818	0.1331
	SCF + DCF	0.0833	0.0469	0.06	0.24	0.091
	SCF + CDF	0.0442	0.0288	0.0345	0.069	0.1089
	SCF + FTF	0	0	0	0	0.0852
	SCF + PSF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3346
	SCF + DCF + CDF + FTF + PSF	0.7258	0.178	0.2787	0.3784	**0.3584**
Decision Tree	SCF	0.0817	0.0576	0.0657	0.15	0.0681
	SCF + DCF	0.0767	0.0485	0.0593	0.186	0.0677
	SCF + CDF	0.1501	0.1104	0.1222	0.1905	0.0827
	SCF + FTF	0.1348	0.1346	0.1305	0.2553	0.1248
	SCF + PSF	0.3917	0.2176	0.2643	0.4444	0.2146
	SCF + DCF + CDF + FTF + PSF	**0.3903**	**0.2561**	**0.301**	**0.5217**	**0.2613**
SVM	SCF	0.0444	0.11	0.0632	0.2529	0.0828
	SCF + DCF	0.2433	0.152	0.1237	0.2391	0.1335
	SCF + CDF	0.5292	0.1189	0.1915	0.2857	0.1725
	SCF + FTF	0.0312	0.0156	0.0208	0.0833	0.0883
	SCF + PSF	**1**	0.2176	0.3508	0.4706	0.2869
	SCF + DCF + CDF + FTF + PSF	0.8239	**0.339**	**0.4678**	**0.5946**	**0.4111**

between class distributions. Since we employ majority voting to classify samples, the combination of high dimensionality and the scarcity of positive samples in both training and test sets adversely affects performance. Conversely, Logistic Regression (LR) yields the best results in terms of both recall and F1 scores. One of the key advantages of LR is its ability to generalize well on small to moderate-sized datasets, particularly when classes can be separated by a linear boundary. Decision Trees may overfit in small datasets, while SVMs can be adversely affected by outliers, especially when using hard margins.

As it is the case with Logistic Regression and SVM, the Decision Tree algorithm appears to give the most successful results in the scenario where all feature sets are present. In addition, Decision Tree offers a transparent and interpretable structure,

Figure 5.5: *Visualization of the Constructed Decision Tree.*



as illustrated in Figure 5.5, making it valuable for understanding feature-based decision logic. In general, incorporation of FTF and PSF (top nodes in the Decision Tree) significantly improves the results.

5.2.2 Performance of ML Algorithms with Feature Combinations Based on Feature Importance

Although we selected SCF as the core feature in the previous section and evaluated the ML algorithm performances, we saw that SCF did not provide sufficient effect as a core feature. Hence, in this section, we analyze the importance of dataset features by adopting the aforementioned feature selection techniques. Consequently, we extract the ranks of each feature based on the feature importance score estimated by the Random Forest as shown in Table 5.5. The correlation analysis results are presented in Table 5.6, and importance scores measured by the select k-best techniques are listed in Table 5.7.

We see that *Previous Build Status* and *Changed Files* stand out across all three

Table 5.5: *Feature Importance Values.*

Feature Category	Feature	Importance
Previous Status Feature	**Previous Build Status**	**0.1916**
File Type Feature	**Changed Files**	**0.1683**
Difference Count Features	Number of modified files	0.0596
Source Code Features	Total Addition Lines	0.0457
Difference Count Features	Number of modified directories	0.0439
Source Code Features	Total Lines	0.0427
Source Code Features	Total Lines Changed	0.0421
Source Code Features	Total Deletion Lines	0.0414
Source Code Features	Android Code Lines	0.0396
Source Code Features	Android Code Deletion Lines	0.0382
Source Code Features	Android Code Lines Changed	0.0366
Source Code Features	Android Code Addition Lines	0.0342
Difference Count Features	Number of modified subsystems	0.0272
Categorized Developer Features	Gender Group	0.0380
Categorized Developer Features	Experience Group	0.0522
Categorized Developer Features	Checking Control Status	0.0398
Categorized Developer Features	Seniority Group	0.0434
Categorized Developer Features	Age Group	0.0378
Difference Count Features	Number of modified apps	0.0219
Source Code Features	Android System Version	0.0160

tables. *Previous Build Status* ranks first in feature importance, followed by *Changed Files* in second place. The remaining features have significantly lower and nearly identical importance scores (see Table 5.5). The same pair of features also emerge as the most prominent in the correlation analysis results shown in Table 5.6. This time, *Changed Files* ranks first, followed by *Previous Build Status* in second place. However, their correlation values are similar and significantly higher than those of the remaining features.

We observe similar results with the normalized score estimated by K-Best feature method, where *Previous Build Status* and *Changed Files* rank the first and second, respectively. Scores attributed to all the other features are significantly lower (See Table 5.7).

Figure 5.6 provides an overview of the results from three feature analysis metrics, Feature Importance, Correlation and Select K-Best, showcasing their evaluation of feature relevance in the dataset. According to these methods, all feature sets are shown in the bar. As can be seen in the chart, PSF and FTF stand out as the most valuable feature sets.

Table 5.6: *Feature Correlation Values.*

Feature Category	Feature	Correlation
File Type Feature	**Changed Files**	**0.6513**
Previous Status Feature	**Previous Build Status**	**0.4702**
Categorized Developer Features	Seniority Group	0.1580
Categorized Developer Features	Experience Group	0.1290
Categorized Developer Features	Checking Group	0.1228
Categorized Developer Features	Age Group	0.1056
Categorized Developer Features	Gender Group	0.0919
Source Code Features	Total Lines	0.0406
Source Code Features	Android Code Lines	0.0386
Difference Count Features	Number of modified apps	0.0322
Difference Count Features	Number of modified subsystems	0.0299
Difference Count Features	Number of modified files	0.0261
Difference Count Features	Number of modified directories	0.0230
Source Code Features	Total Deletion Lines	0.0192
Source Code Features	Android System Version	-0.0188
Source Code Features	Android Code Deletion Lines	0.0167
Source Code Features	Android Code Lines Changed	0.0093
Source Code Features	Total Lines Changed	0.0053
Source Code Features	Android Code Addition Lines	0.0032
Source Code Features	Total Addition Lines	-0.0027

These feature sets are followed by CDF and SCF, respectively. DCF feature set showed relatively worse performance in feature analysis compared to other feature sets.

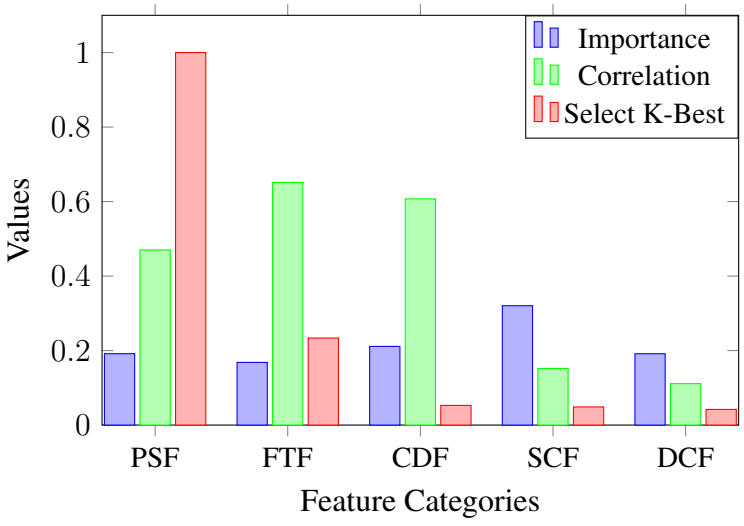
Accordingly, we evaluate ML algorithms using the combinations of most important features, guided by the results of our feature importance analysis. PSF, identified as the most important feature, is selected as the core feature. To systematically assess the impact of feature combinations, we start with PSF alone and incrementally add the other most important feature sets, FTF and CDF. Finally, the SCF feature set, representing the source code metrics in 5.2.1, is also included in the best-performing feature set combination.

Table 5.8 lists the results obtained with traditional ML algorithms when PSF is used as the core feature in feature set combinations. The best performance is generally obtained from the PSF+FTF+CDF feature set combination, where the top three feature

Table 5.7: *Features based on Select K-Best with Min-Max Normalized Scores.*

Feature Category	Feature	Score	Normalized Score
Previous Status Feature	**Previous Build Status**	**518.8367**	**1.000000**
File Type Feature	**Changed Files**	**121.2420**	**0.233668**
Categorized Developer Features	Age Group	16.0419	0.030920
Source Code Features	Total Lines	9.5657	0.018435
Source Code Features	Android Code Lines	9.1613	0.017655
Difference Count Features	Number of modified subsystems	7.1259	0.013731
Categorized Developer Features	Seniority Group	6.3079	0.012152
Difference Count Features	Number of modified directories	6.0608	0.011683
Difference Count Features	Number of modified apps	4.9352	0.009510
Difference Count Features	Number of modified files	4.2045	0.008101
Categorized Developer Features	Checking Control Status	3.6674	0.007066
Source Code Features	Android System Version	2.7057	0.005215
Source Code Features	Total Deletion Lines	1.8081	0.003485
Source Code Features	Android Code Deletion Lines	1.2314	0.002373
Categorized Developer Features	Gender Group	0.8146	0.001570
Source Code Features	Android Code Lines Changed	0.4347	0.000838
Source Code Features	Total Lines Changed	0.2524	0.000487
Source Code Features	Android Code Addition Lines	0.0762	0.000147
Categorized Developer Features	Experience Group	0.0579	0.000112
Source Code Features	Total Addition Lines	0.0001	0.000000

Figure 5.6: *Feature Importance, Correlation, and Select K-Best Scores by Feature Categories.*



sets are used together. Except for Logistic Regression, the inclusion of the SCF feature set did not lead to a significant improvement in performance. Although the performances are generally high, the best results were obtained when all feature sets are used

together (See Table 5.4).

Table 5.8: *Performance Metrics for Traditional Algorithms with Best Feature Sets.*

Algorithm	Feature Set	Precision	Recall	F1	F1 Max	PR-AUC
Logistic Regression	PSF	0.75	0.0591	0.1059	0.2727	0.2707
	PSF + FTF	**1**	0.202	0.3246	0.4706	0.3014
	PSF + CDF	**1**	0.1888	0.3138	0.4375	0.2776
	PSF + FTF + CDF	0.9097	0.2529	**0.3894**	0.5143	0.349
	PSF + FTF + SCF	**1**	0.202	0.3246	0.4706	0.3014
	PSF + FTF + CDF + SCF	0.7917	**0.2625**	0.3834	**0.5556**	**0.3504**
KNN	PSF	0.5439	**0.2176**	0.3032	0.4571	**0.3441**
	PSF + FTF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3065
	PSF + CDF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3378
	PSF + FTF + CDF	**1**	**0.2176**	**0.3508**	**0.4706**	**0.3441**
	PSF + FTF + SCF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3065
	PSF + FTF + CDF + SCF	**1**	0.1976	0.3247	**0.4706**	0.2976
Decision Tree	PSF	0.3083	0.2273	0.2607	0.4091	0.1936
	PSF + FTF	0.2338	0.2273	0.2292	0.375	0.1957
	PSF + CDF	0.3718	**0.2685**	**0.2957**	0.4324	0.2342
	PSF + FTF + CDF	**0.3971**	0.2273	0.2811	0.383	**0.2393**
	PSF + FTF + SCF	0.3729	0.2273	0.2793	**0.439**	**0.2655**
	PSF + FTF + CDF + SCF	**0.3971**	0.2273	0.2811	0.3830	**0.2393**
SVM	PSF	0	0	0	0	0.2824
	PSF + FTF	**1**	0.2176	0.3508	0.4706	0.2985
	PSF + CDF	**1**	0.2176	0.3508	0.4706	**0.3285**
	PSF + FTF + CDF	0.7875	**0.2373**	**0.3626**	**0.5143**	0.3108
	PSF + FTF + SCF	**1**	0.2176	0.3508	0.4706	0.2985
	PSF + FTF + CDF + SCF	0.7875	**0.2373**	**0.3626**	**0.5143**	0.3108

5.3 Effectiveness of Ensemble Algorithms (RQ3)

This section investigates whether the obtained prediction performance could be increased by leveraging ensemble algorithms. For this purpose, we employed all feature set combinations previously used with the traditional algorithms. That is, the two feature set combination approaches used for traditional algorithms were also applied to ensemble algorithms. The first approach follows prior literature by using source code features as the core set. The second approach is guided by feature importance analysis, where PSF is identified as the most important and as such the core feature.

Table 5.9: *Performance Metrics for Ensemble Algorithms and Feature Sets Combined based on Their Adoption in the Literature.*

Algorithm	Feature Set	Precision	Recall	F1	F1 Max	PR-AUC
Random Forest	SCF	0.027	0.0256	0.0259	0.0571	0.0505
	SCF + DCF	0	0	0	0	0.0649
	SCF + CDF	0	0	0	0	0.0778
	SCF + FTF	0.15	0.03	0.05	0.2	0.0702
	SCF + PSF	0.9444	**0.2045**	**0.3238**	**0.4706**	0.2695
	SCF + DCF + CDF + FTF + PSF	0.9167	0.1945	0.3147	**0.4706**	**0.2873**
Gradient Boosting	SCF	0.1227	**0.3837**	0.1476	0.2951	0.2383
	SCF + DCF	0.1565	0.1856	0.1493	0.2642	0.1319
	SCF + CDF	0.1369	0.0901	0.1051	0.3404	0.0875
	SCF + FTF	0.1427	0.0996	0.1098	0.3077	0.0885
	SCF + PSF	**0.5167**	0.2273	**0.2943**	0.5000	0.2590
	SCF + DCF + CDF + FTF + PSF	0.3386	0.2657	0.2936	**0.5098**	**0.2847**
XGBoost	SCF	0.6458	0.058	0.103	0.1429	0.1065
	SCF + DCF	0.25	0.0192	0.0357	0.1429	0.07
	SCF + CDF	0.125	0.01	0.0185	0.0741	0.0664
	SCF + FTF	0.125	0.0192	0.0333	0.1333	0.083
	SCF + PSF	**1**	0.1913	**0.3076**	0.4706	0.3008
	SCF + DCF + CDF + FTF + PSF	**1**	**0.1909**	0.3059	**0.5143**	**0.3032**
LightGBM	SCF	0.2014	**0.0549**	**0.0857**	0.125	0.0671
	SCF + DCF	0.1857	0.0424	0.0682	0.129	0.0702
	SCF + CDF	**0.375**	0.0196	0.037	0.0741	0.0597
	SCF + FTF	0.3438	0.0396	0.064	**0.1818**	0.0679
	SCF + PSF	0.2014	**0.0549**	**0.0857**	0.125	0.0671
	SCF + DCF + CDF + FTF + PSF	0.25	0.0192	0.0357	0.1429	**0.0766**
AdaBoost	SCF	0	0	0	0	0.0922
	SCF + DCF	0	0	0	0	0.0832
	SCF + CDF	0	0	0	0	0.0736
	SCF + FTF	0	0	0	0	0.0901
	SCF + PSF	**1**	**0.2176**	**0.3508**	**0.4706**	**0.6269**
	SCF + DCF + CDF + FTF + PSF	**1**	**0.2176**	**0.3508**	**0.4706**	**0.6269**

Accordingly, Table 5.9 presents the results obtained using ensemble algorithms, applied to the same feature combinations evaluated in the previous section. We do not observe substantial improvements in performance relative to traditional algorithms. The impact of different feature combinations is similar in general. PSF along with SCF appears to be the most influential feature pairs. Moreover, using all feature sets together is

not always advantageous. In all algorithms except for AdaBoost and XGBoost, performance tends to decrease when combining these feature sets. Those algorithms may fail due to the complexity of the input space. AdaBoost and XGBoost are not affected by this issue; they demonstrate consistent performance whether using the SCF and PSF feature sets together or when all feature sets are combined.

An individual analysis of the ensemble boosting algorithms reveals that Adaboost showed the best performance by a considerable margin in F1 score and PR-AUC values. XGBoost and Gradient Boosting can reach upto F1 value of 0.51. Gradient Boosting accompanied Adaboost in recall with the highest value. LightGBM showed the lowest performance among boosting algorithms in all metrics. It can be seen that LightGBM has difficulty learning on unbalanced, difficult and limited datasets. An interesting observation is that both Random Forest and AdaBoost performed extremely poorly in terms of precision and recall without PSF involved in the feature set, with both metrics dropping to zero.

The Gradient Boosting classifier demonstrated an unusually high recall (0.3837) when trained solely on the SCF (Source Code Features), despite low precision and F1 scores. This suggests that the model tended to label a large portion of samples as positive, resulting in a higher number of true positives being detected. One possible explanation is that some SCF features—such as the number of changed lines or modifications specific to Android code—may have incidentally matched patterns found in faulty builds during training. Because Gradient Boosting builds its decision function in a sequential manner and is highly sensitive to subtle feature interactions, it might have captured these coincidental signals too strongly. This may lead the model to predict more positive cases, thus increasing recall, but it may also increase the number of false positives. Consequently, since the hyperparameter tuning process focuses solely on maximizing the F1 score, the resulting model have unintentionally favored higher recall.

Table 5.10 lists the results obtained with ensemble algorithms when feature sets are combined based on their importance. Hereby, PSF is selected as the core feature, providing a foundation to examine how other feature sets influence the results.

Table 5.10: *Performance Metrics for Ensemble Algorithms with Feature Sets Combined based on Their Importance.*

Algorithm	Feature Set	Precision	Recall	F1	F1 Max	PR-AUC
Random Forest	PSF	0.9444	0.2045	0.3238	**0.4706**	0.2695
	PSF + FTF	**1**	0.1164	0.2085	0.2222	0.2729
	PSF + CDF	**1**	**0.2045**	**0.3302**	**0.4706**	0.2803
	PSF + FTF + CDF	**1**	0.1445	0.2401	**0.4706**	**0.2997**
	PSF + FTF + SCF	**1**	0.1945	0.3176	**0.4706**	0.2698
	PSF + FTF + CDF + SCF	**1**	0.1445	0.2401	**0.4706**	**0.2997**
Gradient Boosting	PSF	0.3234	0.2041	0.2467	0.4091	0.1579
	PSF + FTF	0.5455	0.2076	0.2872	0.3721	0.2440
	PSF + CDF	0.6597	0.2176	0.3224	0.4571	**0.2697**
	PSF + FTF + CDF	**0.9410**	0.2045	**0.3235**	0.4571	0.2544
	PSF + FTF + SCF	0.6637	0.1937	0.2737	0.4651	0.2320
	PSF + FTF + CDF + SCF	0.3424	**0.3331**	0.2852	**0.4878**	0.2582
XGBoost	PSF	**1**	0.1913	0.3076	0.4706	0.3008
	PSF + FTF	**1**	0.1913	0.3076	0.4706	0.2952
	PSF + CDF	**1**	0.2176	0.3508	0.4706	0.3132
	PSF + FTF + CDF	0.9444	0.198	0.3205	0.4	0.2922
	PSF + FTF + SCF	**1**	0.1913	0.3076	0.4706	0.2952
	PSF + FTF + CDF + SCF	**1**	**0.2273**	**0.3617**	**0.5143**	**0.3215**
LightGBM	PSF	0.2014	**0.0549**	**0.0857**	**0.125**	0.0671
	PSF + FTF	0.093	0.0488	0.0619	0.1081	0.0499
	PSF + CDF	0.05	0.0132	0.0208	0.0833	0.0528
	PSF + FTF + CDF	0.0625	0.0132	0.0217	0.087	0.0591
	PSF + FTF + SCF	**0.3438**	0.0396	0.064	0.1818	**0.0679**
	PSF + FTF + CDF + SCF	0.2500	0.0096	0.0185	0.0741	0.0635
AdaBoost	PSF	**1**	**0.2176**	**0.3508**	**0.4706**	**0.6269**
	PSF + FTF	1	0.2176	0.3508	0.4706	0.6269
	PSF + CDF	1	0.2176	0.3508	0.4706	0.6269
	PSF + FTF + CDF	1	0.2176	0.3508	0.4706	0.6269
	PSF + FTF + SCF	1	0.2176	0.3508	0.4706	0.6269
	PSF + FTF + CDF + SCF	1	0.2176	0.3508	0.4706	0.6269

In the table where PSF serves as the core, the best performance is generally not obtained from the rows with the most feature sets, unlike the results listed in Table 5.8 and in all ensemble algorithms except LightGBM, the best performance was achieved by

using PSF and CDF together. We also observe that no matter which feature set is added to PSF in AdaBoost, no change in performance is observed.

5.4 Threats to Validity

Our empirical evaluation is subject to an internal validity threat since the observed outcomes can be attributed to external factors that are not captured by features. Projects and developers change over time. We included data from multiple projects to reduce project-specific bias. We also applied consistent data collection procedures and utilize feature-based representations (categorizations) of developers and files to mitigate this risk. Nevertheless, unobserved confounding variables can exist.

An external validity threat arises from the fact that all projects in our dataset are from a specific domain, namely Android-based TV software. Although the dataset includes multiple projects, the findings may not generalize to other industries, technologies, or application domains.

Our approach focuses on predicting the build status after a collection of commits rather than after each individual commit. As a result, the size of the dataset is significantly smaller compared to studies based on single commits. This introduces a conclusion validity threat regarding the inferences drawn about the relationships between feature sets, ML algorithms, and prediction outcomes. To mitigate this risk, we employed varying test sets and reported multiple evaluation metrics to provide a comprehensive assessment of model performance.

6. CONCLUSION AND FUTURE WORK

In this study, we investigated the application of machine learning techniques to predict build outcomes in a Continuous Integration (CI) environment within an industrial setting. Our goal was to identify builds that could be safely skipped to reduce CI costs without significantly increasing the risk of undetected failures. We proposed a prediction approach based on collective code changes contributed by multiple developers and extracted resilient features through the categorization of developers and modified files.

Through comprehensive experiments involving various machine learning algorithms and feature combinations, we observed that ensemble-based algorithms do not significantly outperform traditional models. Among the traditional algorithms, Logistic Regression (LR) yielded the best performance. In terms of feature importance, the previous build status was found to be the most informative. Our results demonstrate high precision but low recall in predicting failed builds, leading to F1 scores of up to 60% when all features are used with LR. While such F1 scores might initially seem low for deployment in an industrial context, the approach remains valuable: it allows hundreds of successful builds to be correctly predicted and skipped, with only a small number of failures potentially being missed. Overall, predicting build status based on collective changes is a challenging task, especially due to class imbalance and small sample size. Nevertheless, it presents a promising opportunity for optimizing CI processes in industrial environments.

As future work, it would be interesting to leverage time-based learning models such as Long short-term memory (LSTM) to capture the temporal relations in data. One can also replicate the same experiments for other projects in various application domains.

REFERENCES

[1] A. K. Arani, M. Zahedi, T. H. M. Le, and M. A. Babar, "Sok: Machine learning for continuous integration," in *2023 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*, pp. 8–13, IEEE, 2023.

[2] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017.

[3] I.-C. Donca, O. P. Stan, M. Misaros, D. Gota, and L. Miclea, "Method for continuous integration and deployment using a pipeline generator for agile software projects," *Sensors*, vol. 22, no. 12, p. 4637, 2022.

[4] N. Singh, "Ci/cd pipeline for web applications," *International Journal for Research in Applied Science and Engineering Technology*, vol. 11, no. 5, 2023.

[5] T. M. Suhas and S. N. K, "Continuous integration and continuous deployment with jenkins in c++ software development," *Journal of University of Shanghai for Science and Technology*, vol. 23, pp. 805–810, 2021.

[6] S. Arachchi and I. Perera, "Continuous integration and continuous delivery pipeline automation for agile software project management," pp. 156–161, 2018.

[7] D. Stahl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.

[8] D. Ståhl, T. Mårtensson, and J. Bosch, "The continuity of continuous integration: Correlations and consequences," *Journal of Systems and Software*, vol. 127, pp. 150–167, 2017.

[9] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella, "A tale of ci build failures: An open source and a financial organization perspective," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 183–193, IEEE, 2017.

[10] Z. Xie and M. Li, "Cutting the software building efforts in continuous integration by semi-supervised online auc optimization.," in *IJCAI*, pp. 2875–2881, 2018.

[11] D. Marijan and S. Sen, "Devops enhancement with continuous test optimization.," in *SEKE*, pp. 536–535, 2018.

[12] M. Kawalerowicz and L. Madeyski, "Continuous build outcome prediction: an experimental evaluation and acceptance modelling," *Applied Intelligence*, vol. 53, no. 8, pp. 8673–8692, 2023.

[13] Y. Hong, C. Tantithamthavorn, J. Pasuksmit, P. Thongtanunam, A. Friedman, X. Zhao, and A. Krasikov, "Practitioners' challenges and perceptions of CI build failure predictions at Atlassian," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 370–381, 2024.

[14] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 211–222, 2015.

[15] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be ci skipped?," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 448–463, 2019.

[16] A. Sharif, D. Marijan, and M. Liaaen, "Deeporder: Deep learning for test case prioritization in continuous integration testing," pp. 525–534, 2021.

[17] D. Marijan, "Comparative study of machine learning test case prioritization for continuous integration testing," *Software Quality Journal*, vol. 31, no. 4, pp. 1415–1438, 2023.

[18] P. Liang, "Automating the training and deployment of models in mlops by integrating systems with machine learning," *Applied and Computational Engineering*, vol. 67, no. 1–7, 2024.

[19] N. G. Camacho, "Unlocking the potential of ai/ml in devsecops: effective strategies and optimal practices," *Journal of Artificial Intelligence General science (JAIGS) ISSN: 3006-4023*, vol. 3, no. 1, pp. 106–115, 2024.

[20] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 233–242, IEEE, 2017.

[21] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2022.

[22] R. Abdalkareem, S. Mujahid, and E. Shihab, "A machine learning approach to improve the detection of ci skip commits," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2740–2754, 2020.

[23] S. Lee, S. Hong, J. Yi, T. Kim, C.-J. Kim, and S. Yoo, "Classifying false positive static checker alarms in continuous integration using convolutional neural networks," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 391–401, IEEE, 2019.

[24] B. Chen, L. Chen, C. Zhang, and X. Peng, "Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 42–53, 2020.

[25] I. Saidani, A. Ouni, and M. W. Mkaouer, "Improving the prediction of continuous integration build failures using deep learning. automated software engg. 29, 1 (may 2022), 61 pages," *Automated Software Engineering*, 2022.

[26] C. Pan and M. Pradel, "Continuous test suite failure prediction," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 553–565, 2021.

[27] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "Branch coverage prediction in automated testing," *Journal of Software: Evolution and Process*, vol. 31, no. 9, p. e2158, 2019.

[28] G. Silva, C. Bezerra, A. Uchôa, and I. Machado, "What factors affect the build failures correction time? a multi-project study," in *Proceedings of the 17th Brazilian Symposium on Software Components, Architectures, and Reuse*, pp. 41–50, 2023.

[29] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 197–207, 2017.

[30] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 41–50, IEEE, 2014.

[31] J. Benjamin and J. Mathew, "Enhancing continuous integration predictions: a hybrid lstm-gru deep learning framework with evolved dbso algorithm," *Computing*, vol. 107, no. 1, p. 9, 2025.

[32] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pp. 13–25, 2020.

[33] X. Jin and F. Servant, "Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration," *Journal of Systems and Software*, vol. 188, p. 111292, 2022.

[34] D. M. Kamath, E. Fernandes, B. Adams, and A. E. Hassan, "On combining commit grouping and build skip prediction to reduce redundant continuous integration activity," *Empirical Software Engineering*, vol. 29, no. 6, p. 145, 2024.

[35] A. Kola-Olawuyi, N. R. Weeraddana, and M. Nagappan, "The impact of code ownership of devops artefacts on the outcome of devops ci builds," in *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 543–555, 2024.

[36] S. Shimmi and M. Rahimi, "On association of code change types and ci build failures in software repositories," *European Journal of Information Technologies and Computer Science*, vol. 4, no. 2, pp. 1–15, 2024.

[37] H. Sozer and C. Gebizli, "Model-based testing of digital TVs: An industry-as-laboratory approach," *Software Quality Journal*, vol. 25, no. 4, pp. 1185–1202, 2017.

[38] F. Charte, A. Rivera, M. J. del Jesus, and F. Herrera, "On the impact of dataset complexity and sampling strategy in multilabel classifiers performance," pp. 500–511, 2016.

[39] M. K. Dahouda and I. Joe, "A deep-learned embedding technique for categorical features encoding," *IEEE Access*, vol. 9, pp. 114381–114391, 2021.

[40] F. Pargent, F. Pfisterer, J. Thomas, and B. Bischl, "Regularized target encoding outperforms traditional methods in supervised machine learning with high cardinality features," *Computational Statistics*, vol. 37, pp. 2671–2692, 2022.

[41] M. Linares-Vásquez, C. McMillan, D. Poshyvanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *Empirical Software Engineering*, vol. 19, pp. 582–618, 2014.

[42] E. Bisong, E. Tran, and O. Baysal, "Built to last or built too fast? evaluating prediction models for build times," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 487–490, IEEE, 2017.

[43] M. Chelvan and K. Perumal, "On the relationship between feature selection stability and change in statistical properties of datasets," *International Journal of Computer Application*, vol. 7, pp. 46–56, 2017.

[44] F. Fabris, A. Doherty, D. Palmer, J. P. De Magalhaes, and A. A. Freitas, "A new approach for interpreting random forest models and its application to the biology of ageing," *Bioinformatics*, vol. 34, no. 14, pp. 2449–2456, 2018.

[45] E. Shi, L. Sun, J. Xu, and S. Zhang, "Multilabel feature selection using mutual information and ml-relieff for multilabel classification," *IEEE Access*, vol. 8, pp. 145381–145400, 2020.

[46] J. Xia and Y. Li, "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)*, pp. 311–315, IEEE, 2017.

[47] X. Wang, Y. Yan, and X. Ma, "Feature selection method based on differential correlation information entropy," *Neural Processing Letters*, vol. 52, pp. 1339–1358, 2020.

[48] N. R. Abid-Althaqafi and H. A. Alsalamah, "The effect of feature selection on the accuracy of x-platform user credibility detection with supervised machine learning," *Electronics*, vol. 13, p. 205, 2024.

[49] K. Zhang, W. Geng, and S. Zhang, "Network-based logistic regression integration method for biomarker identification," *BMC systems biology*, vol. 12, pp. 113–122, 2018.

[50] O. Chamorro-Atalaya, N. Alvarado-Bravo, F. Aldana-Trejo, C. Poma-García, C. Aliaga-Valdez, G. Peralta-Eugenio, and A. A. T. Jala, "Supervised learning through k-nearest neighbor, used in the prediction of university teaching performance," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 27, p. 1625, 2022.

[51] L. Qadrini, H. Hikmah, E. Tande, I. Presda, A. A. Maghfirah, N. Nilawati, and H. Handayani, "Ensemble resampling support vector machine, multinomial regression to multiclass imbalanced data," *BAREKENG: Jurnal Ilmu Matematika Dan Terapan*, vol. 18, pp. 0269–0280, 2024.

[52] W. F. W. Yaacob, S. A. M. Nasir, W. F. W. Yaacob, and N. M. Sobri, "Supervised data mining approach for predicting student performance," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 16, no. 3, pp. 1584–1592, 2019.

[53] P. Das, A. K. Sadhu, A. Konar, B. S. Bhattacharya, and A. K. Nagar, "Adaptive parameterized adaboost algorithm with application in eeg motor imagery classification," pp. 1–8, 2015.

[54] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, *et al.*, "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 13, no. 2, p. e1484, 2023.

[55] F. S. Gomiasti, W. Warto, E. Kartikadarma, J. Gondohanindijo, and D. R. I. M. Setiadi, "Enhancing lung cancer classification effectiveness through hyperparameter-tuned support vector machine," *Journal of Computing Theories and Applications*, vol. 1, pp. 396–406, 2024.

APPENDICES

APPENDIX A. BUILDING TIMES PER PROJECT

Table A.1: *The List of Projects and their Build Completion Times.*

Project	Building Time	Extra Building Time (Extra APKs and OEMs)	Total Software Build Time
P1	2 hours 36 minutes	0 minutes	2 hours 36 minutes
P2	2 hours 45 minutes	28 minutes	3 hours 13 minutes
P3	2 hours 45 minutes	6 minutes	2 hours 51 minutes
P4	2 hours 32 minutes	7 minutes	3 hours 39 minutes
P6	2 hours 22 minutes	9 minutes	2 hours 31 minutes
P5	2 hours 31 minutes	8 minutes	2 hours 39 minutes
P7	2 hours 28 minutes	5 minutes	2 hours 33 minutes
P8	2 hours 22 minutes	0 minutes	2 hours 22 minutes
P9	2 hours 18 minutes	0 minutes	2 hours 18 minutes
Total	**22 hours 39 minutes**	**1 hour 3 minutes**	**23 hours 42 minutes**