Commit Artifact Preserving Build Prediction

Guoqing Wang
guoqingwang@stu.pku.edu.cn
Key Lab of HCST (PKU), MOE;
School of Computer Science, Peking University
Beijing, China

Zeyu Sun*
zeyu.zys@gmail.com
National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences
Beijing, China

Yizhou Chen, Yifan Zhao, Qingyuan Liang
{yizhouchen,zhaoyifan,liangqy}@stu.pku.edu.cn
Key Lab of HCST (PKU), MOE;
School of Computer Science, Peking University
Beijing, China

Dan Hao*
haodan@pku.edu.cn
School of Electronic and Computer Engineering, Peking University
Shenzhen, China

ABSTRACT

In Continuous Integration (CI), accurate build prediction is crucial for minimizing development costs and enhancing efficiency. However, existing build prediction methods, typically based on predefined rules or machine learning classifiers employing feature engineering, have been constrained by their limited ability to fully capture the intricate details of commit artifacts, such as code change and commit messages. These artifacts are critical for understanding the commit under a build but have been inadequately utilized in existing approaches. To address this problem, we propose GitSense, a Transformer-based model specifically designed to incorporate the rich and complex information contained within commit artifacts for the first. GitSense employs an advanced textual encoder with built-in sliding window text samplers for textual features and a statistical feature encoder for extracted statistical features. This innovative approach allows for a comprehensive analysis of lengthy and intricate commit artifacts, surpassing the capabilities of traditional methods. We conduct comprehensive experiments to compare GitSense with five state-of-the-art build prediction models, Longformer, and ChatGPT. The experimental results show that GitSense outperforms these models in predicting failed builds, evidenced by 32.7%-872.1.0% better on F1-score, 23.9%-437.5% better on Precision, and 40.2%-1396.0% better on Recall.

CCS CONCEPTS

• **Software and its engineering** → *Collaboration in software development*; *Software maintenance tools.*

KEYWORDS

Continuous Integration, Deep Learning, Build Prediction

*Corresponding authors: Zeyu Sun and Dan Hao.

ISSTA 2024, 16-20 September, 2024, Vienna, Austria

ACM ISBN 978-1-4503-XXXX-X/24/09…$15.00
https://doi.org/10.1145/3650212.3680356

ACM Reference Format:
Guoqing Wang, Zeyu Sun, Yizhou Chen, Yifan Zhao, Qingyuan Liang, and Dan Hao. 2024. Commit Artifact Preserving Build Prediction. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680356

1 INTRODUCTION

Continuous Integration (CI) is a widely used practice in modern software development, with which developers commit their code to a shared repository in large-scale software development activities [18, 25]. In CI, each commit triggers an automated build on a CI build platform when integrated into the project. The purpose of timely and frequent build in CI is to expose defects earlier, ensure the quality of the project, and improve development efficiency [18, 36, 37, 52, 61]. However, such build actions may also incur very high cost [37]. As reported by previous work, over 40% of the build process consumes more than 30 minutes individually [29], and the frequent build process has cost Google millions of dollars [37]. To one's disappointment, the high cost sometimes cannot yield corresponding benefits. Recent studies [13, 69] have shown that more than 70% of the builds on the CI dataset collected from GitHub are actually passed (i.e., executed without raising any errors). Such a high passing rate suggests the presence of a large proportion of unnecessary build actions that can be skipped, considering that the goal of build in CI is to reveal problems early [38]. Thus, developers in industry skip builds to minimize development costs and enhance efficiency [2]. Additionally, Travis CI developers are asked to provide an advanced mechanism to automatically CI skip specific commits [31]. Abdalkareem et al. [2] conduct a survey of 392 projects in TravisTorrent. As of 2019, at least 58 projects use CI skip. And they survey 40 developers and the results show that 75 percent of the developers believe it will be nice, important, or very important to have a technique that automatically indicates CI skip commits. Hence, researchers propose the task of build prediction in CI, which aims to predict the outcome (i.e., passed or failed) of each build. Build prediction can help developers run fewer builds without missing failing builds and thus help save the CI build cost [38].

The early build prediction attempts are mainly made based on some predefined rules [2], e.g., these approaches skip the builds only making changes in source comments, formatting, non-source

```
// Commit Messgae:
// Remove specific versions for Maven plugins.
// The current versions are a bit outdated. In particular,
// maven-compiler-plugin is pinned at version 3.1, which has a known bug
// with incremental compilation, causing all code to be compiled in every
// Maven build. Unfortunately it hasn't been fixed yet. Maven 3.x will pick
// the latest version automatically if no version is specified for the plugins.

// Changed files:
pom.xml
pom-main-shaded.xml
pom-gwt.xml
```

```
1.      <artifactId>maven-bundle-plugin</artifactId>
2. -    <version>2.4.0</version>
3.  </plugin>
            ......
4. -    <version>1.7</version>
            ......
5.      <attifactId>gwt-maven-plugin</artifactId>
6. -    <version>1.0-rc-4</version>
            ......
7.      <artifactId>maven-surefine-plugin</artfactId>
8. -    <version>2.17</version>
```

Figure 1: Commit *d85b7bc* in Closure.

files, meta-files, or version-release files. However, these manually predefined rules are hard to encompass the diverse and complex scenarios encountered in CI. Consequently, this approach tends to yield limited performance and is prone to having false positives (i.e., failed builds that should not be skipped but are mistakenly skipped) [39]. To address these problems, machine-learning-based approaches have been introduced [1, 3, 13, 38, 41, 53, 54], resulting in enhanced performance. Instead of designing rules manually, these approaches leverage machine learning techniques to build a prediction model which takes features reflecting the commits' characteristics as input. However, these features are still manually extracted from the project repository, build history, and commit under a build.

Despite feature engineering advancements, existing machine-learning-based approaches have not achieved satisfactory performance on build prediction due to their inherent limitations in manual feature extraction. That is, manual feature extraction can hardly catch all the features relevant to commits' characteristics and thus some valuable features are ignored, e.g., code change in the commit and commit message that reflects the purpose of the commit. Such information lies in the artifact of a commit, but is seldom used by the existing machine-learning-based approaches. Additionally, there is usually no clear indication to illustrate directly whether it can be skipped, such as a "Skip Build" flag. A similar phenomenon has also been reported by Abdalkareem et al. [2]. Hence, this information must be utilized according to the semantics. Figure 1 shows a commit of *d85b7bc* in the closure-compiler project. In this commit, the developer only changes the configuration files by mistakenly removing specific versions for Maven plugins, addressed in red in the figure. Due to the changes, the corresponding build is failed. However, the existing rule-based and machine-learning-based approaches predict it as a passed build because the corresponding commit does not contain source code change. We notice that the commit message "maven-compiler-plugin is pinned at version 3.1, which has a known bug...hasn't been fixed yet" may imply a possible build failure. Unfortunately, this information is not recognized in the rules or manually-extracted features of the existing approaches.

However, it is challenging to leverage the artifact of a commit in build prediction. An ideal way is to use neural networks to encode the artifact as textual inputs, such as changed code and commit messages. However, the practicality of this method is hampered by the typically lengthy and intricate nature of input commits. Our statistics indicate that over 60% of code changes within commits exceed 1,000 tokens in length. This presents a significant challenge, as existing neural models like CodeBERT [23] are limited to handling inputs of less than 512 code tokens, due to the memory constraints. Applying these models in their current form would necessitate the omission of a substantial amount of semantic information from the inputs. Furthermore, neural networks generally struggle to accurately extract features that require statistical analysis from the inputs [10]. For example, the previous build state and the failed ratio of the previous five builds which reflect the previous build history are the most important features for previous machine-learning-based approaches [13, 38, 41]. These features are hard to extract from the inputs solely based on the neural networks. Additionally, the statistical values on code changes are also important (e.g., the number of dependencies that are added to the new code and the number of modified fields). These kinds of statistical values are also challenging to be accurately extracted from the code inputs.

To address these problems, we propose a Transformer-based commit artifact preserving build prediction model, GitSense. To encode lengthy textual inputs, GitSense employs multiple sets of sliding window text samplers. These samplers are designed to capture the overall features of lengthy textual inputs while maintaining low memory usage. Moreover, for the precise extraction of features that require statistical analysis, GitSense incorporates a statistical tool to comprehensively extract relevant statistical features. Specifically, GitSense traces the project's commit history and obtains statistical features from the build history and the code repository. The statistical information includes the features of the current build, the features of the previous build, and the features of the build histories. Subsequently, GitSense introduces a feature encoder module, dedicated to encoding these features. Finally, GitSense integrates these encoded features to predict results.

The experiment was conducted on an existing dataset, which is constructed by Chen et al. [13] and contains 20 projects. We use the initials of the first three authors of the paper to refer to this dataset, CCZ. We compare our proposed approach with five state-of-the-art approaches (i.e. SmartBuildSkip [38], BuildFast [13], HYBRIDCISAVE [41], MOGP [53], and TF [3]). Considering that there are some variants of transformer architecture developed for processing long inputs, we further include Longformer [8] in our evaluation. Besides, we also compare our proposed approach against ChatGPT [48], because it has achieved promising results in some code-related fields (e.g., code generation [46, 71] and automated program repair [58, 60, 65]). The experimental results show that 1) GitSense outperforms the seven approaches for predicting failed builds. Specifically, GitSense is 32.7%-872.1% better on F1-score, 23.9%-437.5% better on Precision, and 40.2%-1396.0% better on Recall than these existing approaches; 2) GitSense achieves these

enhanced prediction results without time expenditure on processing long texts. The prediction time per sample for GitSense is 0.017 seconds. This result is comparable to other machine-learning-based approaches, whose prediction times per sample range between 0.001 seconds and 0.271 seconds. Compared with Longformer and GPT which need 0.092 and 1.910 seconds to predict, our model achieves much less time cost; 3) all the modules designed in GitSense contribute to the overall effectiveness. 4) GitSense can be combined with existing machine-learning-based approaches to further enhance their effectiveness. We integrate GitSense into the two compared approaches BuildFast and HYBRIDCISAVE, and find that the integration improves their F1-score from 0.294 to 0.650, and from 0.195 to 0.664, respectively.

To sum up, this paper makes the following contributions.

- We propose GitSense, a novel commit artifact preserving build prediction model. GitSense can process 50,000 tokens at once and is adept at encoding not only textual information but also the features extracted from the repository and build histories. To the best of our knowledge, we are the first to introduce textual inputs into the build predictions;
- We extend a previous dataset and conduct a comprehensive evaluation on this extended dataset, CCZ. The technique implementation, training code, and evaluation data are available at our website [49];

2 APPROACH

2.1 Overview

The architecture of GitSense is shown in Figure 2. GitSense contains four modules: 1) input extraction module, which extracts both textual and statistical features from the given commit; 2) feature selection module, which selects important statistical features from the original statistical features related to the commits; 3) encoder module, which uses a Transformer to encode both the textual features and the selected important statistical features for subsequent processing; 4) prediction module, which uses a set of dense layers to predict the build skip results. In the following subsections, we will introduce each module of GitSense in detail.

2.2 Input Extraction

The input extraction phase aims to get the inputs used for GitSense. Our approach considers two types of inputs: (1) textual features, which are derived from the artifact of the commit under build, and (2) statistical features, which are obtained from the build history and the code repository.

The textual features include the corresponding commit message, the name of the changed files, and the changed code. In particular, we extract the commit message from the commit message repository according to the unique commit ID and capture file names across various file formats. The changed files are defined as the differences between two commits, encompassing additions, deletions, modifications, and other edit operations in the file. Our statistics are based on the calculation of changes using the GitHub API [30]. For changed code, we extract the changed code through the GitHub commit logs [30]. To preserve contextual information about code change, we explicitly enhance change representation by adding "+" or "-" flags before the code to indicate code additions or code deletions. For unchanged context information, we incorporate it without any flag. We represent the textual features of a commit as $T_O =< C_C, C_M, C_N >$, where T_O refers to the commit under build, C_C refers to its changed code, C_M refers to its commit message, and C_N refers to the name of changed files.

In our approach, we reuse the statistical features proposed in previous work [1, 7, 13, 39, 54]. In particular, our statistical feature list includes the features of the current build, the features of the previous build, and the features of the build histories. These statistical features can be summarized as the number of added/deleted/changed lines, fields, classes, imports, source files, and test files in the current build; the corresponding developers' information in the current build; the build state and changes of the previous build; the developer team in the project; the success or fail probabilities of the whole repository at multiple code granularity. All these features to be analyzed are denoted as $F =< f_1, f_2, \cdots, f_K >$, where K is the dimension of the features.

2.3 Feature Selection

Feature selection aims to filter all the acquired statistical features, and extract the important ones from them. According to previous studies [1, 39, 40], some features are critical for prediction, while some features may even have a negative side effect on classification.

To tackle such a problem, we propose to exclude useless features from the feature list. For these statistical features, GitSense uses an XGBoost classifier to conduct an initial prediction in the training dataset. We select XGBoost due to its good performance in classification tasks [14, 32]. In XGBoost, our input consists of all extracted statistical features, and the output is the result of build prediction, where 1 represents passed build (i.e. can be skipped), and 0 represents failed build (i.e. cannot be skipped). We do not focus on the classification results in this process, instead, we leverage XGBoost to assess the importance of features in the classification process. When the importance of each statistical feature is obtained, we sort the statistical features by importance and extract top-M features from them. We represent the set of extracted features as $F_i =< f_{i_1}, \cdots, f_{i_j}, \cdots, f_{i_M} >$, where M is the dimension of the important statistical features. That is, we keep the statistical features that are important for prediction and discard the other statistical features. In our work, the dimension of selected important statistical features M is 32, which is the balance between effectiveness and efficiency. As for how to select the optimal number of features for building a prediction model, we are willing to leave it for future work to explore based on our architecture. The list of important statistical features and brief descriptions are shown in Table 1.

2.4 Input Encoder Module

This module aims to encode complex textual features and statistical features as embedded vectors. To avoid confusing inputs in representation, we use two different encoder modules to learn representations for textual features and statistical features, named textual encoder and statistical feature encoder. We will introduce the two encoders separately.

2.4.1 Textual Encoder. The main intention of the textual encoder module is to extract the important information in the long textual inputs and achieve dimension reduction in data representation.

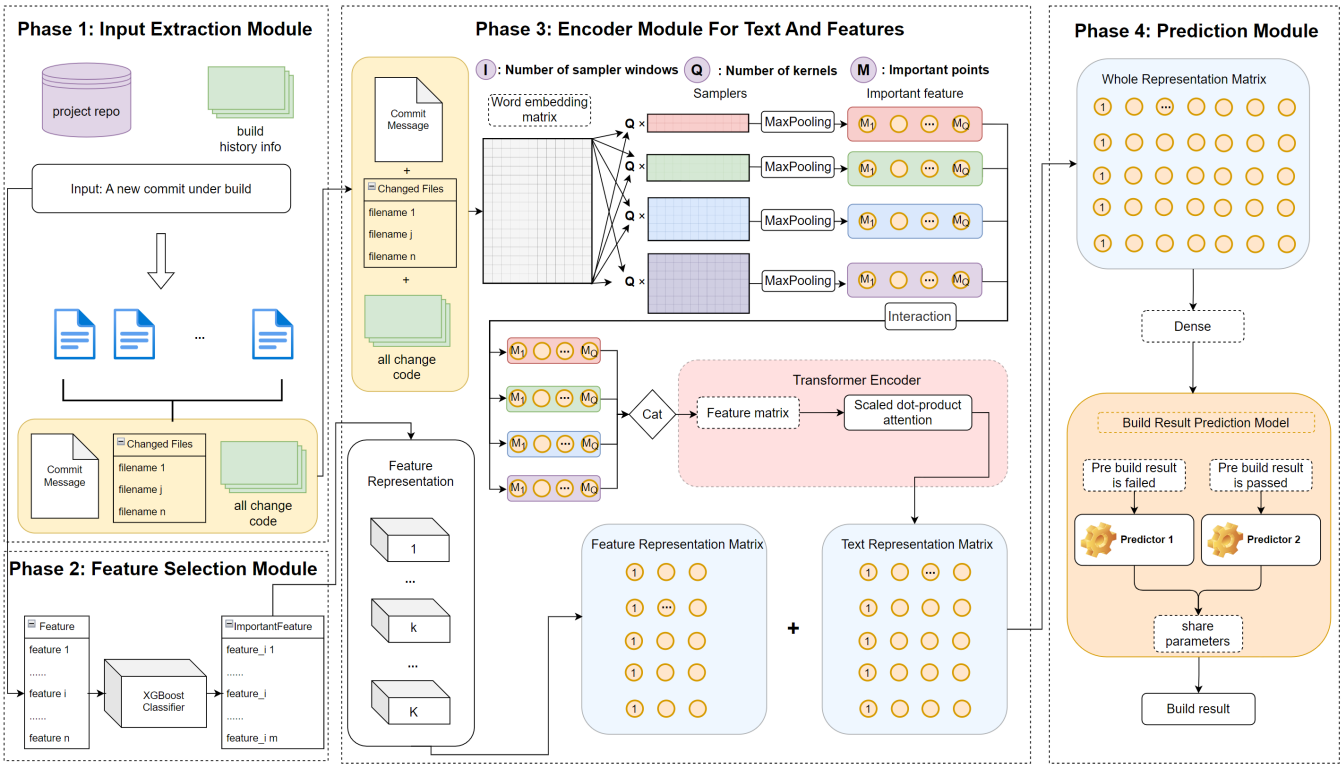

Figure 2: The Detailed Process of GitSense.

Table 1: The Important Statistical Features

Feature	Description	Feature	Description
last_label	last build result	src_files	the production files changed
fail_ratio_pr	the ratio of broken builds in all the previous builds	methodbody	the number of method bodies modified
log_src_files_in	size of the intersection of log_src_files and src_files	log_src_files	the number of production files reported in the build log of the previous build
pr_test_exception	whether tests throw exceptions	add_import	the number of import statements added
pr_src_files	production files changed between the latest passed build and the previous build	gh_team_size	the size of team contributing in last 3 months
fieldchange	fields modified, added or deleted	files_added	the number of files added
last_fail_gap_avg	builds since the last broken build is	classchange	the number of classes modified, added or deleted
prev_srcchurn	lines of production code changed of previous build	deletemethod	the number of methods deleted
commits_on_files	commits on the files in last 3 months	test_ast_diff	whether test code is changed in AST
pr_tests_ok	the number of tests passed	deletefieldchange	the number of fields deleted
file_fail_prob_sum	sum of the probability of each changed file involved in previous broken builds	merge_commits	the number of merge commits included
fail_ratio_com_re	broken builds in recent 5 builds that were triggered by the current committer	src_churn	the number of lines of production code changed
consec_fail_builds_sum	the sum of consecutive broken builds	line_deleted	the number of deleted lines in all files
file_fail_prob_max	the maximum of the probability of each changed file involved in previous broken builds	commits	the number of commits included
pr_duration	overall time duration of the build	is_master	whether the build occurs on master branch
by_core_member	whether a core member triggers the build	line_added	the number of added lines in all files

Given a commit T_O and its commit message C_M, the name of changed files C_N, and the code change C_C, we first concatenate them together as $[C_C; C_M; C_N]$. We tokenize it into a token sequence $T_T = \{t_1, t_2, \cdots, t_N\}$ via the *longformer tokenizer* [8], where N denotes the number of the tokens. These tokens are represented as real-valued vectors $T_E = \{\boldsymbol{t}_1, \boldsymbol{t}_2, \cdots, \boldsymbol{t}_N\}$ via *embeddings*.

To encode these long tokens, inspired by an existing approach [15], we use a set of reader windows I with different lengths in the CNNs. These windows are applied across the whole elements in T_E. The goal is to sample representations with various dimensions, which can capture diverse information on the sampled representations. Notably, to extract a wide range of non-redundant features, we use Q convolution kernels within each window, which are all initialized with unique parameters of the same shape. These kernels aim to explore different blocks of code snippets and extract the key contents during the training phase. For each kernel, this process can be represented as follows:

$$C_m^{i,q} = \sum W^{i,q} \cdot T_{E_{m:m+l_i-1}} + b$$

where $W^{i,q} \in \mathbb{R}^{l \times k}$ denotes the $q-$th convolution kernel of the $i-$th ($i \in I$) window with the length l_i, and C_m denotes the representation of the $m-$th fragment of T_E by $W^{q,i}$, b is a bias. k represents the size of embedding dimensions.

All the representations of the fragments sampled by the convolution kernel are then concatenated into a whole representation $C_q^i = [C_1^{i,q}, C_2^{i,q}, \cdots, C_{N-l_i+1}^{i,q}]$. Then a max pooling is utilized to compress $(N-l_i+1)$-dimensional representation into 1-dimensional.

$$M_q^i = \max(C_q^i)$$

The maximum weight of all Q kernels in specific sampled windows are concatenated into a new representation matrix $M^i = [M_1^i, M_2^i, \cdots, M_Q^i]$. Finally, the textual encoder module combines different representation matrices M by I sampled windows as the output O, which represents approximately important semantic contents of the whole textual input, denoted as $O \in \mathbb{R}^{I \times Q}$:

$$O = [M^1, M^2, \cdots, M^I]$$

In this way, the multi-step sampled vectors provide a condensed representation of the whole long textual inputs, which filters unimportant blocks of code and enables subsequent processing to focus on the most important and informative parts of the code. Additionally, this characterization reduces the computational overhead of transformer [62] and only requires one traversal of the vector to find the maximum value.

After that, we utilize the multi-head self-attention mechanism which is widely used in the language models [8, 23, 62] to promote information interaction among the vectors in O. This facilitates the efficient capturing of contextual dependencies within the different parts of the long text inputs. It can be described as follows:

$$head_s = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

where $head_s$ is the output of the $s-$th attention head. The query Q, key K, and value V are computed by different weight matrices. The outputs of all heads are further jointed together with a weight matrix W, which can be represented as follows:

$$P_t = [head_1; \cdots; head_H]W$$

2.4.2 Statistical Feature Encoder. For these important statistical features, GitSense needs to represent them as an embedding vector for the neural network to understand the meaning of the features. Hence, we utilize two dense layers to provide an abstract representation of the statistical features in a high-dimensional space. The subsequent equation illustrates the computation process of these independent important features F_i.

$$P_f = F_i W^T + \boldsymbol{b}$$

where W is the weight matrix, $\boldsymbol{b}$ is the bias vectors, and P_f is the output of the feature encoder module.

The respective representations through the textual encoder and statistical feature encoder modules can be represented as P_t and P_f, respectively. Considering the non-interoperability of statistical features and textual features, we turn the two parts into a final representation $P = [P_t; P_f]$ by stitching them together for further handling.

2.5 Prediction Module

Considering that successive failures are a common occurrence in building systems [13, 38, 41], we utilize two sub-predictive models in our prediction module. The first one predicts the samples that the previous build is failed, and the other one predicts the samples that the previous build is passed. Notably, instead of dividing the two sub-predictive models into two separate parts, GitSense share parameters between the two sub-models in order to maximize the performance and generalization capabilities of the model. A similar process is also utilized in the encoder module to enhance the representation capability.

$$Result = \begin{cases} MLP_f(P), & \text{if } P_{pre} \text{is failed} \\ MLP_p(P), & \text{if } P_{pre} \text{is passed} \end{cases}$$

where P presents the representation of the inputs and P_{pre} presents the result of the previous build. MLP_f and MLP_p present the two sub-predictive models, i.e., two multi-layer perceptrons for predicting the samples that the previous build is failed and the previous build is passed, respectively. Both of the two models use softmax as the activation function for the last layer to produce a classification probability with the range of $[0, 1]$.

GitSense (i.e., the build prediction task) is optimized via the following objective:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)]$$

where N is the number of samples, y_i is the true label of the i-th sample, taking values of 0 or 1, p_i is the predicted probability by the model that the i-th sample belongs to the positive class.

3 EXPERIMENTAL SETUP

In this section, we describe the dataset, baselines, evaluation metrics, and experimental results. Our evaluation is designed to answer the following research questions.

RQ1. *How does GitSense perform compared with the state-of-the-art baselines in predicting build results?* To answer this question, we conduct experiments on an established dataset and compare GitSense with four state-of-the-art approaches.

RQ2. *What is the contribution of each component in GitSense?* To answer RQ2, we analyze GitSense, systematically removing each component to discern its specific contribution to the overall effectiveness of GitSense.

RQ3. *Whether the textual encoder module in GitSense can be transferable to improve other models?* The textual encoder module is the key component of GitSense. To answer RQ3, we show the impact of the textual encoder module by integrating it, along with the prediction modules, into other machine-learning-based models.

3.1 Dataset

To evaluate the effectiveness of our proposed method, we utilize the build prediction dataset extracted from TravisTorrent, CCZ, provided by Chen et al. [13] [1], containing 20 GitHub high-star Java projects. Unfortunately, no build prediction dataset includes text information only contains features extracted from the origin build systems. We use a reverse mining approach [30] based on the extracted constructed features, labels, and build ids to locate the original commits under the building action. Then, we extract the

[1]They overcome the issues of TravisTorrent, such as missing build details, unattainable build features, and some projects being too small [2, 13], and construct this dataset we used, enhancing it from the TravisTorrent source.

Table 2: The Dataset Statistics

Statistics	Number
Passed Builds	23,912
Failed Builds	5,159
Sum of Changed Files	1,085,358
Average Changed Files of Per Build	37.33
Sum of Added Lines	53,141,763
Average Added Lines of Per Build	1828.0
Sum of Deleted Lines	30,966,137
Average Deleted Lines of Per Build	1065.2

commit messages, changed files list, and code change compared with the previous version, respectively. In the tracing phase, we cannot succeed in tracing the IWicket-Bootstrap project's build systems, and find its detailed build and commit corresponding histories. Thus, we remove this project from our used dataset. Hence, our dataset contains 19 projects selected by Chen et al. and extracted by us. We present the statistics about these projects in Table 2, including the sum number of passed and failed builds and the average number of extracted text tokens under build. Following previous work [1, 13, 38], we split the builds into the training and testing datasets by 3:1 for each project. Notable, to simulate the practical usage scenario, we partition the dataset in chronological order. As previous work reports, the performance of existing techniques obtained by widely-used cross-validation deviates from the performance in practical scenarios [13, 66]. In the setup of chronological order, the prediction of a build's outcome relies solely on the knowledge acquired from previous builds, which can minimize deviations. After splitting, there are 21,813 samples in our training dataset with 17,798 samples of passed builds and 4,015 samples of failed builds. The testing dataset contains 7,258 samples with 6,114 samples of passed builds and 1,144 samples of failed builds.

3.2 Baselines

In our experiment, we compare with state-of-the-art build prediction approaches, including SmartBuildSkip [38], BuildFast [13], HYBRIDCISAVE [41], MOGP [53], and TF [3]. Considering that there are some variants of transformer that target handling long inputs, we include Longformer [8] in our evaluation as a representative. Additionally, with the LLMs (large language model) spreading, ChatGPT [48] has achieved much attention and great progress in many fields, especially in the code-related field. Hence, we also utilize ChatGPT to predict build outcomes.

SmartBuildSkip. SmartBuildSkip differentiates between first failures and subsequent failures. It first uses a machine-learning classifier to predict build outcomes to capture first failures with the features describing the current build and the project. To maximize cost savings, after observing a first failure, it determines all subsequent builds fail until it observes a build outcome pass. It then uses the machine-learning classifier to predict build outcomes.

BuildFast. BuildFast is a history-aware approach to predict CI build outcomes cost-efficiently and practically. It extracts multiple failure-specific features from closely related historical builds via analyzing build logs, the current build, and the build repository's history information. The features after purification are fed to the XGBoost classifier to predict the build outcome.

HYBRIDCISAVE. HYBRIDCISAVE combines the prediction of multiple techniques to obtain stronger certainty with a Random Forest classifier. Compared with other build prediction approaches, it also considers the test information and the dependency between the tests and source code to predict which builds to skip fully and which builds to skip partial test executions.

MOGP. MOGP introduces a search-based approach based on multi-objective genetic programming to build a prediction model. The selected features include ten types, such as code information and testing information. Compared to previous approaches, it aims to simultaneously consider passed and failed builds to find the optimal combination of features and corresponding thresholds.

TF. TF utilizes textual analysis to measure the frequency count of token appearances in the source code instead of any statistical features extracted from code and repos. It then uses the term token frequency metric as the input of a prediction model. Particularly, TF employs a random forest classifier to predict build outcomes.

Longformer. Longformer utilizes an attention mechanism based on local windowed attention that scales linearly with sequence length. The attention mechanism is a drop-in replacement for the standard self-attention. It outperforms standard transformer models on long document tasks. We use the newer version of Longformer that supports 16K tokens in our evaluation.

ChatGPT. LLMs get much attention due to their remarkable performance in generating and understanding code and texts. Among them, ChatGPT is a popular text content generation product. We include it in our evaluation as a representative of LLMs to comprehensively evaluate the performance in build result prediction.

Because ChatGPT is a zero-shot model, the prediction results are tied to the provided prompt. Hence, in order to achieve a comprehensive and fair comparison, we evaluate it in two ways. The first edition is asking it with all the text inputs. In our prompt, we introduce the CI scenario and explain the meaning of inputs. The second edition is asking it with all the text inputs and also includes our extracted features. For the features, we write a prompt with detailed descriptions of each feature and the corresponding number. The prompt's format is as follows. "*In continuous integration, there are some builds that can be skipped. Now I will give you a build with some texts and features and you need to determine if this build can be skipped.*" The following content is features and texts. The features are the description of every statistical feature and the corresponding number. The texts are the textual features of the commit. The two kinds of prompts can be achieved in our reproducible packages.

3.3 Evaluation Metrics

To measure the performance of our approach, we follow previous studies [1, 13, 16, 38, 39, 41, 54] and use some standard metrics in our evaluation, such as precision, recall, and F1-score. The metrics are defined as follows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}} \in [0,1] \quad \text{Precision} = \frac{\text{TP}}{\text{TP+FP}} \in [0,1]$$

where TP is the number of failed builds that are classified as failed; FN is the number of failed builds that are classified as passed; FP denotes the number of passed builds that are identified as failed.

However, in many fields, recall and accuracy are high in one and low in the other. In order to synthesize the effects of prediction, we introduce the F1-score, which is the reconciled average of precision and recall. It can be represented as follows:

$$\text{F1-score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \in [0, 1]$$

Considering that build prediction is a cost-sensitive task, developers hope to skip as many passed builds as possible. However, if the prediction is not correct, developers may work on the buggy code base and need to redo or roll back their work. In this situation, those integration errors may accumulate for a long time without timely correction, increasing the fixing efforts, which may cause developers to pay much more costs. Hence, as previous studies report, we introduce Gain in our evaluation as before [13, 38, 39]. For better understanding and comparison among different methods, we further normalize Gain. We first need to calculate the Benefit. It represents the skipped builds' duration. The skipped build duration is the aggregation of the duration of all build steps (e.g., build preparation steps and tests) that a technique decides to skip, for all builds in our dataset. Next, we need to calculate the side effort, i.e. cost, which means the aggregation of the failed builds' duration that a technique decides to skip. All the records of duration are collected from the CI build platform. Gain is the difference between benefit and cost and then divided by the total duration of passed builds. The metric can be denoted as below.

$$\text{Gain} = \frac{\text{Benefit} - \text{Cost}}{\text{all passed build duration}}$$

3.4 Implementation

Our approach is implemented based on PyTorch [51]. In the encoder, the lengths of sampler windows are 2, 3, 5, 7, and 11, respectively. The number of kernels is set up to 200. The maximum length for encoding in our neural network is 50,000. For the hyperparameters, we configure the six-layer transformer with the dropout rate of 0.1 to prevent overfitting and 200-dimension hidden states. Our model is optimized by AdamW [42] with learning rate of 0.00005. We set hyper-parameters for baselines as suggested by their authors.

The experiments are performed on a machine with Intel Xeon CPU E5-2680 v4 @ 2.40GHz, running Ubuntun 16.04.6 LTS. The models are trained on four 24G GPUs of GeForce RTX 3090.

4 RESULTS

4.1 Effectiveness of GitSense (RQ1)

To reduce randomness in the results, we run each method for five times and then report the averaged results. Since each call to the ChatGPT's interface needs to be billed separately, we only tested it for one time on the testing dataset.

The performance results of RQ1 are shown in Table 3. We report the performance of predicting both failed builds and passed builds to compare the performance of different methods comprehensively. The F1-score$_f$ represents the F1-score of predicting failed builds, and the F1-score$_p$ represents the F1-score of predicting passed build in the dataset. Other metrics can be understood in the same way. The ChatGPT$_{text}$ corresponds to the prompts only containing text inputs and the ChatGPT$_{all}$ corresponds to the prompts containing both text inputs and feature inputs.

As shown, for failed builds, we find that GitSense achieves the best performance with 0.661 in the F1-score, 0.602 in Precision, and 0.733 in Recall. Compared with the machine-learning-based approaches (i.e., SmartBuildSkip, BuildFase, HYBRIDCISAVE, MOGP, and TF), SPLICE is 32.7%-872.1% better on F1-score, 23.9%-437.5% better on Precision, and 40.2%-1396.0% better on Recall. While TF predicts builds based on the frequency of tokens, its processing granularity for text features is overly coarse and fails to consider the sequential order of textual features. Consequently, this processing approach remains fundamentally reliant on statistical features. When considering Longformer which can also handle long textual inputs, the attention mechanism of Longformer makes it easier to focus on neighboring tokens but not filter out irrelevant information and capture important parts. This is also the reason that Longformer performs well in the text summarization task. In contrast, our model utilizes a combination of CNN and sliding windows to focus attention only on the most pertinent data, significantly enhancing efficiency for build predictions. We also find that the ChatGPT had the worst results among all the approaches, indicating that the zero-shot model cannot handle the problem of build prediction well. ChatGPT$_{all}$ is even worse than ChatGPT$_{text}$, which shows that even if the features are explained in detail, the input of the features cannot be understood, and has a negative impact on the prediction of the zero-shot large language model. According to the results, all previous models cannot achieve more than 0.5 in Precision and Recall, which indicates that the previous models cannot classify correctly for most failed builds only based on the statistical feature inputs or the textual inputs. GitSense achieves 0.661 of F1-score, which suggests the importance of commit artifacts in predicting whether a build can be skipped precisely.

Considering the effectiveness differences of training strategies, we also compare GitSense with the best two approaches (i.e., SmartBuildSkip and BuildFast) from the baseline under other strategies (i.e., single-project training and testing, and leave-one-out validation). In single-project training strategy, the F1/recall/precision result of GitSense is 0.68/0.66/0.70, higher than SmartBuildSkip (0.59/0.54/0.61) and BuildFast (0.55/0.58/0.56). In leave-one-out validation, the average F1/recall/precision result of our approach is 0.73/0.77/0.67, higher than SmartBuildSkip (0.61/0.64/0.58) and BuildFast (0.42/0.48/0.38). These results demonstrate GitSense's consistent effectiveness across training strategies. Although these two training strategies can produce better results, considering the importance of model generalization, we decide to follow the same training strategy as previous work [1, 3, 12, 13, 36, 53] which can better reflect real-world applications.

We further present the performance of predicting passed builds. GitSense achieves a Precision score of 0.948, only worse than HYBRIDCISAVE, which performs bad in predicting failed builds, and achieves a Recall score of 0.909. When accuracy and recall are considered together, our method also ranks the second in terms of effectiveness for successful builds with only 0.005 worse than HYBRIDCISAVE. Considering that build prediction is an unbalanced classification problem, the number of failed builds is much less than the number of passed builds. When predicting failed builds wrong,

Table 3: Performance comparison of different approaches.

Approach	F1-score$_f$	Precision$_f$	Recall$_f$	F1-score$_p$	Precision$_p$	Recall$_p$	Gain	Prediction Time(sec)
GitSense	**0.661**	**0.602**	**0.733**	0.928	0.948	0.909	**0.865**	0.017
SmartBuildSkip	0.498	0.486	0.523	0.902	0.907	0.896	0.805	0.002
BuildFast	0.294	0.386	0.272	0.919	0.915	0.927	0.811	0.012
HYBRIDCISAVE	0.195	0.131	0.385	**0.933**	**0.973**	0.896	0.826	**0.001**
MOGP	0.143	0.134	0.154	0.908	0.915	0.901	0.793	0.271
TF	0.068	0.112	0.049	0.881	0.839	0.922	0.753	0.002
Longformer	0.054	0.173	0.032	0.880	0.810	0.961	0.776	0.092
ChatGPT$_{all}$	0.003	0.105	0.002	0.913	0.842	**0.997**	0.810	1.910
ChatGPT$_{text}$	0.062	0.115	0.042	0.887	0.840	0.940	0.761	1.910

it may cause the delayed discovery of errors and even greater financial losses. Hence, we should pay more attention to the failed build prediction, which is mentioned in previous work [13, 39].

When considering Gain, our approach GitSense achieves 0.865 of Gain, which is the most in all the approaches. The second one is HYBRIDCISAVE, achieving 0.826 of Gain. The small gap between each method on this metric is due to the high predictive accuracy of each method for passed builds and the small penalty for predicting wrongly. The cost of predicting such a failed build wrongly is far more than the Benefit of subtracting the duration of failed builds from the formula of Gain. In reality, the cost of wrong prediction for failed builds is often very high, and developers need to spend a lot of time rolling back to locate errors and redeveloping code.

The results above suggest that in build prediction, our approach GitSense is more effective than existing state-of-the-art models including the large language model, ChatGPT.

Efficiency Evaluation. According to previous research papers [13, 38], the time overhead of predicting builds is a factor that needs to be taken into account. Hence, we further report the time overhead of the compared approaches in Table 3.

From the table, we can find that the prediction times for all methods are acceptable, and in particular, the prediction times averaged over each build are negligible relative to the time to execute the build (1,683.028 seconds). GitSense takes 0.017 seconds per build. This result is marginally higher compared to most machine-learning-based methods, which typically complete predictions in under 0.012 seconds except for MOGP with a prediction time of 0.271 seconds[2]. The slightly increased time of GitSense can be attributed to its neural network-based architecture, which incorporates text inputs and a text encoder module. These components necessitate additional computational time for processing and generating predictions, yet they contribute to the model's enhanced accuracy and effectiveness in build prediction tasks. However, GitSense still outperforms Longformer, which has a prediction time of 0.092 seconds. This suggests that although Longformer and GitSense all use sliding windows, GitSense's sampling mechanism achieves better efficiency. Additionally, GitSense outperforms ChatGPT, which has a longer prediction time of 1.910 seconds. The extended prediction time of ChatGPT can be attributed to its status as a large language model with an extensive number of parameters, resulting in higher computational overhead.

[2]MOGP was developed using Java, which introduced a lot of jar packages and involved many time-consuming function calls.

Answer to RQ1: GitSense outperforms state-of-the-art approaches (i.e., SmartBuildSkip, BuildFase, HYBRIDCISAVE, MOGP, and TF) as well as Longformer, with improvements ranging from 23.9% to 437.5% in Precision, 40.2% to 1396.0% in Recall, and 32.7% to 872.1% in F1-score for failed builds. Large language models, such as the current version of ChatGPT, do not perform well when predicting builds. When considering efficiency, the prediction time for all methods is acceptable. The average time for GitSense when predicting each build is only 0.017 seconds.

4.2 Ablation Test (RQ2)

In RQ2, we conduct an ablation test to discern the contribution of each component(i.e. the textual feature encoder module, the statistical feature encoder module, and the impact of the sliding window text samples in the textual encoder module) of our model. The ablation results are presented in Table 4. GitSense$_{W/Ofeature}$ represents GitSense without the statistical feature encoder module, GitSense$_{W/Otext}$ represents GitSense without the textual feature encoder module, and GitSense$_{W/Owindow}$ represents the replacement of the sliding window text samples in the textual feature encoder module of GitSense with a standard Transformer [62] module. Due to the high computational complexity (i.e. n^2) of Transformers, in our experimental setup, a maximum of 1000 tokens can be encoded. For portions of textual features exceeding 1000 tokens, we conventionally apply the truncation processing [23].

The results demonstrate that each component plays a role in enhancing the overall performance of GitSense. When the statistical feature encoder module is removed from GitSense, the model GitSense$_{W/Ofeature}$ can only predict using textual information. The performance metrics for each aspect decrease compared to the complete GitSense, especially for Precision. Specifically, the Precision$_f$ of GitSense$_{W/Ofeature}$ decreases from 0.602 to 0.530, the Recall$_f$ decreases from 0.733 to 0.706, and the F1-score$_f$ decreases from 0.661 to 0.606. For predicting passed builds, the performance of GitSense$_{W/Ofeature}$ has a slight decrease compared to GitSense as well. This suggests that features of statistical analysis play a crucial role in accurately predicting builds that can be safely skipped. Conversely, when the text encoder module is removed

Table 4: The ablation test on GitSense, where GitSense denotes the origin mode of our approach.

Approach	F1-score$_f$	Precision$_f$	Recall$_f$	F1-score$_p$	Precision$_p$	Recall$_p$	Gain
GitSense	**0.661**	0.602	**0.733**	0.928	**0.948**	0.909	**0.865**
GitSense$_{W/Ofeature}$	0.606	0.530	0.706	0.911	0.941	0.883	0.828
GitSense$_{W/Otext}$	0.627	**0.634**	0.621	**0.931**	0.930	**0.932**	0.862
GitSense$_{W/Owindow}$	0.610	0.596	0.626	0.925	0.929	0.921	0.851

from GitSense, the model GitSense$_{W/Otext}$ relies solely on features of statistical analysis. Under the circumstance, the Recall$_f$ experiences a significant decrease compared to the complete GitSense, indicating that textual information is crucial for accurately predicting builds that cannot be skipped. Specifically, the Recall$_f$ decreases from 0.733 to 0.621, and the F1-score$_f$ decreases from 0.661 to 0.627. For predicting passed builds, the performance of GitSense$_{W/Otext}$ has a slight increase compared to GitSense. The improved performance metrics (e.g. F1 and recall for passed builds) stem from the unbalanced nature of build predictions, where passed builds far outnumber failed ones. A lack of sufficient textual features for identifying failures can lead GitSense to inaccurately predict builds as passing. However, the decrease in effectiveness for predicting failed builds cannot be ignored given the huge impact of predicting wrongly failed builds. Incorrect predictions of failed builds, as highlighted before, can delay error identification and potentially lead to greater financial losses. When the set of sliding window text samples is removed from the textual encoder module, we utilize a standard Transformer [62] module to complete encoder textual features. For predicting failed builds, the performance metrics for each aspect of GitSense$_{W/Owindow}$ decrease compared to the complete GitSense, especially for Recall. Specifically, the Recall$_f$ of GitSense$_{W/Owindow}$ decreases from 0.733 to 0.626, the Precision$_f$ decreases from 0.602 to 0.596, and the F1-score$_f$ decreases from 0.661 to 0.610. This decrease demonstrates the importance of incorporating the entire textual features into the textual encoder for comprehensive analysis. Otherwise, the model would be unable to capture the information hidden within complex textual features that lead to build failure. This aligns with our motivation examples, where the key information for predicting failed builds is predominantly embedded in the texts.

Answer to RQ2: Each component plays an important role in enhancing the overall performance of GitSense. After removing the feature encoder module, the Precision$_f$ decreases by 11.96% (i.e., from 0.602 to 0.530). After removing the textual encoder module, the Recall$_f$ decreases by 15.28% (i.e., from 0.733 to 0.621). When replacing the sliding window text samples, the F1-score decreases by 7.72% (i.e., from 0.661 to 0.610).

4.3 Transferability of the Textual Encoder Module of GitSense (RQ3)

In our Introduction, we highlighted the difficulties existing machine-learning approaches face in capturing all relevant features of a commit, especially those within commit artifacts. To address this, we introduced a textual encoder in GitSense that better interprets the nuanced information in these artifacts.

In RQ3, we aim to explore the textual encoder's impact within GitSense further. We integrate our textual encoder with previous machine-learning models to enhance their performance, maintaining our prediction module to combine the strengths of both approaches effectively. Specifically, we replace our feature module's inputs with the features selected by previous models. We combine our model with two best-performing BuildFast and HYBRIDCISAVE, denoted as BuildFast$_{GitSense}$ and HYBRIDCISAVE$_{GitSense}$, respectively. The results are presented in Table 5.

For predicting builds that are failed, both BuildFast$_{GitSense}$ and HYBRIDCISAVE$_{GitSense}$ outperforms the the original machine-learning-based models. For BuildFast$_{GitSense}$, the scores of the F1-score, Precision, and Recall in predicting failed builds improve by 121.1%, 49.5%, and 174.3%, respectively. Furthermore, the score of the F1-score in predicting passed builds also improves by 0.4% than before, from 0.919 to 0.923. A similar effect enhancement is reflected in HYBRIDCISAVE$_{GitSense}$. The score of the F1-score, Precision, and Recall in predicting failed builds improves by 240.5%, 397.7%, and 75.8%, respectively. Additionally, the score of the F1-score in predicting passed builds also improves by 0.3% than before.

Such results suggest that the textual encoder can be effectively integrated with other methods, and plays an important role in build prediction. It indicates the significant impact of the textual information on enhancing the effectiveness of predicting failed builds. Additionally, the architecture design of the text encoder module and the hybrid prediction module exhibits good generalizability in build prediction. Moreover, the improvement in the effectiveness of integrating other models does not come at the cost of losing accurate predictions for successful builds.

Answer to RQ3: The textual encoder module of GitSense improves the performance of the state-of-the-art approaches by integrating with them, such as BuildFast and HYBRIDCISAVE by 121.1%-240.5% in the F1-score for failed builds. There is also a slight improvement in the prediction of successful builds. The results demonstrate the importance of textual information and the generalizability of our textual encoder module.

5 CASE STUDY

In this section, we further present a real-world example to introduce the effectiveness of GitSense.

Figure 3 shows a commit of $571e11b$ in the quickml project. We present the commit message, the changed files, the changed code,

Table 5: The performance improvement of previous approaches by integrating with GitSense.

Approach	F1-score$_f$	Precision$_f$	Recall$_f$	F1-score$_p$	Precision$_p$	Recall$_p$	Gain
BuildFast	0.294	0.386	0.272	0.919	0.915	0.927	0.811
BuildFast$_{GitSense}$	0.650 ↑	0.577 ↑	0.746 ↑	0.923 ↑	0.950 ↑	0.898	0.850 ↑
HYBRIDCISAVE	0.195	0.131	0.385	0.933	0.973	0.896	0.826
HYBRIDCISAVE$_{GitSense}$	0.664 ↑	0.652 ↑	0.677 ↑	0.936 ↑	0.939	0.932 ↑	0.872 ↑

the build result, and the outcome predicted by GitSense, BuildFast, HYBRIDCISAVE, and ChatGPT. In this commit, the developer modifies the buildTree method in the TreeBuilder class to improve readability, deletions addressed in blue, and additions addressed in green in the figure. The change involves 1 file, with 36 additions and 74 deletions. Additionally, the previous five builds are all passed. However, the commit finally triggers a failed build caused by the wrong deletion of the original code. The developer even makes several subsequent changes to fix the failed build soon after. For this complex optimizing change, a necessary build must be conducted. For the compared state-of-the-art machine-learning-based methods, their predictions are only based on the statistical features, such as the previous build result and the minor changes to projects. Unfortunately, this textual information that can reflect the build failure is not recognized in the existing machine-learning-based approaches. In the absence of comprehensive textual information, these models are misled by these features, and they all predict the example to skip. With the textual encoder module's comprehensive understanding of textual information, our proposed approach, GitSense predicts this example to not skip successfully.

The example suggests that, while predicting the build results of certain commits, the commit textual inputs can provide extensive information that cannot be revealed by feature engineering. Furthermore, our proposed model, GitSense not only can handle long inputs and extract important textual information but also can combine textual inputs and features in build prediction.

// **Commit Messgae:**
// greatly simplified the growTree (formerly known as buildTree) method in the
// TreeBuilder

// **Changed files:**
src/main/java/quickml/supervised/classifier/decisionTree/TreeBuilder.java

```
1.      import quickml.supervised.classifier.decisionTree.scorers.GiniImpurityScorer;
2.  -   import quickml.supervised.classifier.decisionTree.scorers.MSEScorer;
3.      …
4.  +   Map<String, AttributeCharacteristics> attributeCharacteristics;
5.      …
6.  -   for (final T instance : trainingData) {
7.  -       for (final Entry<String, Serializable> attributeEntry :
        instance.getAttributes().entrySet()) {
8.  +   int samplesToSkipPerStep = Math.max(1, trainingData.size() /
                RESERVOIR_SIZE);
9.  +   for (int i=0; i<numSamples; i+=samplesToSkipPerStep) {
10. +       for (final Entry<String, Serializable> attributeEntry :
    trainingData.get(i).getAttributes().entrySet()) {
11.     …
12. -   if (trueTrainingSet.size() < this.minLeafInstances) {
13. -       return thisLeaf;
14. -   }
15. -   …
```

// **Build Result:**
Ground Truth: Failure

// **Prediction Outcome:**

GitSense:	Not Skip
BuildFast:	Skip
HYBRIDCISAVE:	Skip
ChatGPT$_{all}$:	Skip
ChatGPT$_{text}$:	Skip

Figure 3: Commit $571e11b$ in quickml.

6 THREATS TO VALIDITY

The threats to internal validity mainly lie in implementing compared techniques of our experiments. To mitigate this threat, we directly use the released code of these methods except for TF without its method code. We implement the method according to its description in the paper since its reproducible package is unavailable. Note that our implementation has achieved consistent results with their reported results. Another threat may appear in the prompt choice of ChatGPT. Different prompts can lead to different outcomes. To alleviate this threat, we refer to prior work [20, 28] and experiment with several different prompts, ultimately selecting the one that yields the best results. Furthermore, the implementation of our model is based on the published model and packages [51, 64] to avoid faults in re-implementation. To further ensure the correct implementation of GitSense, other authors have checked the model.

The threats to external validity lie in the dataset, which may influence the generalization of our findings. To reduce this threat, we use a dataset used in previous work extracted from the popular Travis CI, which has been analyzed by many other research works [1, 13, 38, 39, 41, 54, 69]. All the corresponding text data are collected from the responding GitHub repositories. The projects in the dataset are all Java, which may influence the generalization of our model. We are not sure whether the same results could be achieved on other projects or other programming languages. In future work, we need to further evaluate the effectiveness and efficiency of the scalable data of different programming languages.

Threats to construct validity lie in the metrics we used. All these metrics are reported by developers as describing the value and cost of build prediction in CI and are also used by other existing approaches for saving cost in the build phase of CI.

7 RELATED WORK

We review the most closely related work on build prediction, cost reduction in CI, and empirical studies about CI.

Build Prediction. Hassan and Zhang [34] utilize decision trees to predict build results according to social, technical, coordination, and prior-certification characteristics. Finlay et al. [24] predict build outcomes by adopting data stream mining techniques based

on code-related metrics (i.e., basic metrics, dependency metrics, complexity metrics, cohesion metrics, and Halstead metrics). Xie and Li [67] apply a semi-supervised online AUC optimization to improve the prediction of previous approaches. Abdalkareem et al. [1, 2] study the characteristics of commits in CI, and propose two techniques to automatically identify and predict commits that can be CI skipped. One is based on rules [2] and the other is based on rules and machine learning [1]. Ni and Li [47] and Chen et al. [13] use machine-learning classifiers to predict build results with some specific historical features, such as previous build status and historical development information). Saidani et al. [53] introduce a search-based approach based on multi-objective genetic programming to simultaneously consider pass and failed builds. They collect ten types of features and feed them into a decision tree model. Due to the limitation of machine-learning-based models, they improve their model via a deep-learning classifier [54]. Jin and Servant [38, 40, 41] focus on using machine learning with different features to predict builds, such as features correlated with build information [38] and features with build information and rules [40]. Their newest approach, HYBRIDCISAVE, combines the prediction of multiple techniques to obtain stronger certainty with a Random Forest classifier [41]. Considering the weakness of statistical features, Al-Sabbagh et al. [3] leverage the frequency count of token appearances in the source code to construct a random forest classifier. Although they try to use the textual features, the coarse granularity makes it similar to previous approaches based on the statistical features.

Compared with all previous build prediction techniques (e.g., [1, 2, 24, 34, 54, 67]), our proposed approach GitSense is the first to extract textual information as part of the input to the model.

Cost Reduction in Other Ways. In some situations, the build cannot be skipped completely due to some errors. Hence, there are some studies to explore what phase can be skipped in the complete build process. Some existing techniques target to reduce costs in CI by skipping partial tests within each build [22, 56, 57] or skipping complete test suites [50]. These techniques try to skip tests from different aspects, e.g., tests that correspond to unchanged classes [57] and modules [22, 56, 57], tests that are based in a particular industrial context [45]. Other methods consider skipping tests that fail less historically [35] or applying a machine learning classifier to predict whether tests can be passed [43]. Recently, Elsner et al. [21] propose a framework that can evaluate and compare many regression test selection techniques.

To advance feedback of regression testing in CI, prioritization-based techniques are proposed. Test case prioritization (TCP) in CI aims to prioritize fault-revealing tests ahead and detect faults as early as possible. In particular, existing work on TCP in CI can be classified as empirical studies [9, 21, 33, 39, 68, 70] and TCP techniques [9, 19, 59]. Empirical studies investigate the design decisions of TCP techniques [39], the input of TCP techniques [21, 68], and compare the performance of different categories of TCP techniques [9, 33, 70] to better understand and help improve TCP techniques in CI. TCP techniques in CI use heuristic strategies [19, 33, 44] or machine learning models [4, 9, 11, 17, 55, 59] to predict the priority value of each test and rank them accordingly. Concretely, heuristic-based TCP techniques utilize prior knowledge to design heuristic strategies to assign priority values for tests. For example, Elbaum et al. [19] design formulas to calculate the priority value of a test based on its latest execution timestamp and its recent failure history. In contrast, machine learning-based TCP techniques harness the power of machine learning models to learn prioritization strategies automatically. They have been shown to have better effectiveness and lower overhead compared to heuristic-based TCP techniques [55, 59, 68], which can be further categorized as reinforcement learning-based TCP techniques [4, 17, 59] and supervised learning-based TCP techniques [9, 11, 55] based on their learning paradigms. However, these TCP techniques have also been recognized by some studies as not cost-saving [40, 41].

Considering that there are many other phases for a build in CI except for executing tests, some other existing techniques target to reduce cost by skipping other steps within a build. Celik et al. propose an approach to skip retrieving unnecessary files from within dependencies [12]. Some methods are proposed to skip unnecessary steps when preparing the build environment [26, 27]. The strategy of batching multiple builds [5, 6] or multiple targets [63] together to further reduce cost in CI.

Notable, these techniques are orthogonal to our proposed model GitSense as they focus on different aspects of CI. Hence, ideally, they can be integrated to achieve optimal cost reduction.

8 CONCLUSION

In this paper, we propose a novel build prediction technique, GitSense, which first introduces textual inputs into build prediction. GitSense utilizes a textual encoder module and a statistical feature encoder module to explicitly encode textual features and statistical features. For handling long textual inputs, GitSense employs sliding window text samplers to sample text and extract important features. We perform an extensive evaluation to compare GitSense with five state-of-the-art build prediction techniques on a widely-used benchmark. The results show that GitSense outperforms all compared techniques in terms of F1-score, Precision, and Recall in predicting failed builds. We further analyze the effectiveness of each component in GitSense by an ablation study. We further perform an experiment to evaluate the generalizability of the textual encoder module in GitSense by integrating it into two machine-learning-based models. The results consistently demonstrate the good generalizability of the text encoding module.

9 DATA AVAILABILITY

We make our code, data, and results publicly available in [49].

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No. 62372005 and the Major Project of ISCAS (ISCAS-ZD-202302).

REFERENCES

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2020. A machine learning approach to improve the detection of ci skip commits. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2740–2754.
[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2019. Which commits can be CI skipped? *IEEE Transactions on Software Engineering* 47, 3 (2019), 448–463.
[3] 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*. 42–51.
[4] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2021. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering* (2021).
[5] Amir Hossein Bavand and Peter C Rigby. 2021. Mining historical test failures to dynamically batch tests to save CI resources. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 217–226.
[6] Mohammad Javad Beheshtian, Amir Hossein Bavand, and Peter C Rigby. 2021. Software batch testing to save build test resources and to reduce feedback time. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2784–2801.
[7] 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.
[8] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv:2004.05150* (2020).
[9] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1–12.
[10] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).
[11] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: an industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering*. 975–980.
[12] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build system with lazy retrieval for Java projects. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 643–654.
[13] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. 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*. 42–53.
[14] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.
[15] Yizhou Chen. 2024. Vulnerability-Hunter: An Adaptive Feature Perception Attention Network for Smart Contract Vulnerabilities. *arXiv preprint arXiv:2407.05318* (2024).
[16] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. 2024. Improving Smart Contract Security with Contrastive Learning-based Vulnerability Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 940–940.
[17] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. 2020. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering* (2020).
[18] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
[19] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.
[20] Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. 2023. From commit message generation to history-aware commit message completion. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 723–735.
[21] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically evaluating readily available information for regression test optimization in continuous integration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 491–504.
[22] Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. 2022. Build system aware multi-language regression test selection in continuous integration. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 87–96.
[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[24] Jacqui Finlay, Russel Pears, and Andy M Connor. 2014. Data stream mining for predicting software build outcomes using source code metrics. *Information and Software Technology* 56, 2 (2014), 183–198.
[25] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.
[26] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane Mcintosh. 2020. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering* 48, 6 (2020), 2040–2052.
[27] Alessio Gambi, Zabolotnyi Rostyslav, and Schahram Dustdar. 2015. Poster: Improving cloud-based continuous integration environments. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 797–798.
[28] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with llms?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 761–773.
[29] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* 24 (2019), 2102–2139.
[30] GitHub. 2023. *GitHub REST API Documentation*. https://docs.github.com/en/rest Accessed: 2023-12-12.
[31] GitHub. 2023. *Travis-CI Issues*. https://github.com/travis-ci/travis-ci/issues/6301 Accessed: 2023-12-12.
[32] Aashish Gupta, Shilpa Sharma, Shubham Goyal, and Mamoon Rashid. 2020. Novel xgboost tuned machine learning model for software bug prediction. In *2020 international conference on intelligent engineering and management (ICIEM)*. IEEE, 376–380.
[33] Alireza Haghighatkhah, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98.
[34] 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.
[35] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 483–493.
[36] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 197–207.
[37] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 426–437.
[38] Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 13–25.
[39] Xianhao Jin and Francisco Servant. 2021. What helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 213–225.
[40] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* 188 (2022), 111292.
[41] Xianhao Jin and Francisco Servant. 2023. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–39.
[42] Ilya Loshchilov and Frank Hutter. 2018. Fixing weight decay regularization in adam. (2018).
[43] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 91–100.
[44] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 540–543.
[45] Ricardo Martins, Rui Abreu, Manuel Lopes, and João Nadkarni. 2021. Supervised learning for test suit selection in continuous integration. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 239–246.
[46] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124* (2023).
[47] 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.
[48] OpenAI. 2021. ChatGPT: A Large-Scale Chatbot Model. *OpenAI Blog* (2021). https://www.openai.com/blog/chatgpt
[49] Replication package. 2024. *GitSense*. https://github.com/GuoqingWang1999/GitSense
[50] Cong Pan and Michael Pradel. 2021. Continuous test suite failure prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 553–565.
[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and

Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019). https://pytorch.org/ Accessed: [2023-12-10].

[52] Gustavo Pinto, Marcel Rebouças, and Fernando Castor. 2017. Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users. In *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 74–77.

[53] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology* 128 (2020), 106392.

[54] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering* 29, 1 (2022), 21.

[55] Aizaz Sharif, Dusica Marijan, and Marius Liaaen. 2021. DeepOrder: Deep learning for test case prioritization in continuous integration testing. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 525–534.

[56] August Shi, Suresh Thummalapenta, Shuvendu K Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing test placement for module-level regression testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 689–699.

[57] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and improving regression test selection in continuous integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 228–238.

[58] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[59] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 12–22.

[60] Nigar M Shafiq Surameery and Mohammed Y Shakor. 2023. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290* 3, 01 (2023), 17–22.

[61] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 805–816.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[63] Kaiyuan Wang, Daniel Rall, Greg Tener, Vijay Gullapalli, Xin Huang, and Ahmed Gad. 2021. Smart build targets batching service at Google. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 160–169.

[64] Thomas Wolf et al. 2019. Transformers: State-of-the-Art Natural Language Processing. *arXiv preprint arXiv:1910.03771* (2019).

[65] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[66] 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.

[67] Zheng Xie and Ming Li. 2018. Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization.. In *IJCAI*. 2875–2881.

[68] Ahmadreza Saboor Yaraghi, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2022. Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts. *IEEE Transactions on Software Engineering* (2022).

[69] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 176–187.

[70] Yifan Zhao, Dan Hao, and Lu Zhang. 2023. Revisiting Machine Learning based Test Case Prioritization for Continuous Integration. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 232–244.

[71] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).