What Makes a Good Commit Message?

Yingchen Tian*
Beijing Institute of Technology
Beijing, China
tianyc10@foxmail.com

Yuxia Zhang*†
Beijing Institute of Technology
Beijing, China
yuxiazh@bit.edu.cn

Klaas-Jan Stol
University College Cork and Lero
School of Computer Science and IT
Cork, Ireland
k.stol@ucc.ie

Lin Jiang
Beijing Institute of Technology
Beijing, China
jianglin17@bit.edu.cn

Hui Liu†
Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

ABSTRACT

A key issue in collaborative software development is communication among developers. One modality of communication is a commit message, in which developers describe the changes they make in a repository. As such, commit messages serve as an "audit trail" by which developers can understand how the source code of a project has changed—and why. Hence, the quality of commit messages affects the effectiveness of communication among developers. Commit messages are often of poor quality as developers lack time and motivation to craft a good message. Several automatic approaches have been proposed to generate commit messages. However, these are based on uncurated datasets including considerable proportions of poorly phrased commit messages. In this multi-method study, we first define what constitutes a "good" commit message, and then establish what proportion of commit messages lack information using a sample of almost 1,600 messages from five highly active open source projects. We find that an average of circa 44% of messages could be improved, suggesting the use of uncurated datasets may be a major threat when commit message generators are trained with such data. We also observe that prior work has not considered semantics of commit messages, and there is surprisingly little guidance available for writing good commit messages. To that end, we develop a taxonomy based on recurring patterns in commit messages' expressions. Finally, we investigate whether "good" commit messages can be automatically identified; such automation could prompt developers to write better commit messages.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**; **Programming teams**.

KEYWORDS

Commit-based software development, open collaboration, commit message quality

ACM Reference Format:
Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What Makes a Good Commit Message?. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510205

1 INTRODUCTION

Collaborative software development is an inherently social activity, and is commonly facilitated through version control systems (VCS) such as Git [75, 82]. A VCS maintains a record of code changes, and manages simultaneous access to development artifacts [82]. Each commit should contain both changes to source code (and other stored artifacts) and a message that explains *what* changes are made, and *why* [13, 49]. Figure 1 shows an example of a commit message from the Spring-boot project [64]. A well-written message is needed to communicate the context of a change to collaborators which allows them to review the change and understand its impact [2, 69]. For long-lived projects, such as the Linux kernel, commit messages might be the only source of information left for future developers to understand what changes were made and why those were made, when the original developers have left a project [68].

Software development is increasingly done in distributed settings involving developers from many different cultures and backgrounds. As well, in the past 20 years, commercial participation in open source software (OSS) projects has increased dramatically [77–79, 81], leading to further diversification in the developer workforce on OSS projects that have become essential building blocks for many software organizations. This in turn may further diversify

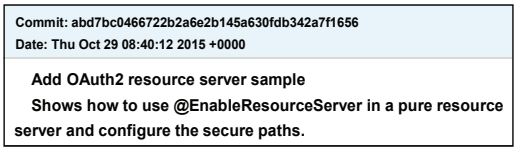

Figure 1: Example of a commit message from Spring-boot

*Yingchen Tian and Yuxia Zhang made equal contributions to this work.
†Corresponding authors

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510205

the quality of commit messages as individual developers and organizations may exhibit different development cultures and habits.

Several researchers have found that the quality of commit messages in repositories varies due to a lack of motivation or time [26, 44, 45, 47]. For example, previous studies observed that ca. 14% of commit messages in over 23,000 OSS projects were completely empty, 66% of the messages contained only a few words, and only 10% of commits had messages containing "normal" descriptive English sentences [26]. Chahal and Saini [14] proposed a syntactic model to calculate the quality of commit messages. However, this model can only assess the quality at the syntactic level through evaluation of "rules," such as *"the first character of the subject line should be capitalized"*; this model does not consider the semantics of commit message contents. Further, developers may not know what kind of information should be written to produce a good commit message [68]. The practitioner community is keen to help their contributors understand how to write a good commit message, as evidenced by guidelines such as: *"the commit message should describe what changes our commit makes to the behavior of the code, not what changed in the code"* [51].

To help developers write commit messages (addressing a potential lack of motivation and time), several tools have been proposed that can generate commit messages automatically [13, 18, 45]. Tools like DeltaDoc [13] and ChangeScribe [18] can produce detailed messages based on the changes contained within a commit, that can mainly answer *what* was changed. However, the generated messages cannot reveal *why* the change was necessary. Inspired by previous observations that commit messages follow certain patterns [49], recent approaches [38, 39, 45] generated commit messages from prior changes and their associated commit messages, which may contain the rationale for any code changes. For those approaches, the quality of generated commit messages relies on messages of similar code changes in the training data. A major problem is that the quality of commit messages in OSS projects used in training datasets for automatic generators might vary considerably. Therefore, using low-quality commit messages in training datasets introduces a major threat if these "poor" commit messages are not filtered out. Moreover, when the generated messages are the same or similar to the low-quality messages in testing datasets, automatic tools may yield an artificially high precision. Unfortunately, many models proposed thus far were trained and evaluated on datasets of commit messages, which simply removed trivial messages without paying attention to the content quality of these messages.

This state of affairs leaves a number of important open questions unanswered. First, (RQ1) to what extent do poorly composed commit messages exist? To the best of our knowledge, no prior work has defined or analyzed what makes a "good" commit message, and subsequently analyzed commit messages to assess the extent of how the quality of messages varies. Second, (RQ2) having established what makes a good commit message at a high level, what are recurring patterns of how these well-written messages are expressed? As we observed, the current state of the art does not consider the semantics of messages, only their syntax. Finally, future tools that could assist developers in writing good commit messages should be able to recognize whether a written message is "good" or not—for example, tools that can prompt developers in real-time as they attempt to make a commit may help improve the quality of commit messages. These tools would also be very useful for researchers in constructing high-quality datasets of commit message generation. Hence, our third research question is (RQ3): can commit messages of good quality be automatically identified?

To answer these questions, we conducted a multi-method study. We first studied and analyzed a set of ca. 1,600 commit messages sampled from five major OSS projects. We defined a "good" commit message as one that explains what was changed, and why a change was made. We found that around 44% of commit messages lack 'why' or 'what' information. This highlights the risk of generating messages based on an unfiltered training dataset that includes low-quality commit messages. Second, we qualitatively analyzed 252 messages with "good" message labels to identify expression characteristics. Third, we built a model based on Bi-LSTM to automatically identify well-written messages, achieving good performance.

This paper makes a number of practical and theoretical contributions to the literature on understanding and identification of high-quality commit messages. Specifically, we 1) propose a set of criteria for identifying well-written messages; 2) demonstrate that considerable proportions of commit messages lack essential information, thus highlighting that this variation in quality requires measures to mitigate; 3) propose a taxonomy of the expressions of commit messages; and 4) build an automated classifier to identify well-written messages.

In the remainder of this paper, we review related work in Sec. 2, outline our multi-method research approach in Sec. 3, and present the results of our study in Sec. 4. We discuss the implications for research and practice in Sec. 5. We present threats to the validity of our reported findings in Sec. 6 and conclude the paper in Sec. 7.

2 RELATED WORK

Commit messages constitute an important modality in collaborative software development for sharing knowledge among developers and in establishing an audit trail of the evolution of a software project. We discuss prior literature that has focused on understanding and utilizing commit messages and how to automatically generate commit messages.

Commit messages are a key resource when addressing several software engineering challenges. One stream of research has focused on classifying code changes into different types by utilizing commit messages manually or automatically to assist maintenance [25, 49, 54]. For example, Mockus and Votta [49] identified three types of commits: adaptive, corrective, and perfective, consistent with Swanson's typology of maintenance activities [67]. Based on the proposed commit types, numerous classification models have been proposed, and commit messages play an important role [25, 42, 74].

A second stream of research has focused on the measurement of quality of code changes by analyzing commit messages. For example, Agrawal et al. [2] studied the evolution of commit quality in five projects by measuring (among others) the number of unique commit messages, and found that the quality of commits declined over time. Santos et al. [58] studied the relationship between "unusual messages" and code quality in commits, and found that unusual messages correlate with build failures, suggesting that these messages serve as a warning sign.

While it is clear that commit messages play an important role in communication among developers, developers may lack time or motivation to craft good commit messages that clearly communicate what is being committed. To address this, several scholars have proposed automatic approaches to automatically generate messages. Some of them are rule-based or use predefined templates [13, 18, 43, 63]. For instance, Buse and Weimer [13] used symbolic execution to generate path predicates between versions of code changes, then populated pre-defined templates and applied summarization transformations to generate commit messages for code changes. Two important limitations of these commit messages generated based on templates are (1) a lack of flexibility, and (2) they cannot convey the intent of committing changes, which only exists in a developer's mind until it is written. Recent studies rely on advanced techniques, such as information retrieval and deep learning [35, 38, 44–46, 50, 72] to generate commit messages automatically. These tend to rely on reusing messages of similar code changes. For example, Huang et al. [36] calculated syntax, semantic, pre-syntax, and pre-semantic similarities of changed code fragments between two versions to find similar code changes and reuse their messages. While specific models vary in their techniques, a common feature is that they take prior commit messages as a key input. For these information retrieval and deep learning based tools, the quality of the manually written commit message is difficult to guarantee [26, 45], which may threaten the effectiveness of these tools.

A few studies investigated the content of commit messages but mainly focused on specific aspects. For example, Alomar et al. explored how developers document their refactoring activities in commit messages, and found that developers tend to explicitly mention the improvement of certain quality attributes and code smells [4]. Chahal and Saini [14] constructed a model that can judge the quality of commit messages by calculating 11 syntactical measures. Text content in other OSS development activities has been studied, such as what/how to document when submitting patches [68] and what information is needed in a bug report [83]. The results of our study on the distribution and expression categories of good commit messages and their relationship with maintenance activities can complement prior understanding of what is a good commit message and how to write one. Moreover, we propose a good-message identification tool that can be used to prompt developers to write better commit messages and build high-quality datasets for the task of automatically generating commit messages.

3 STUDY DESIGN

To address the three research questions introduced in Sec. 1, we conducted a multi-method study. To address RQ1, we compiled a dataset of commits and manually classified the messages based on our definition of a "good" commit message. Sec. 3.1 describes sample selection, data collection, and data processing steps. Sec. 3.2 describes our approach to classifying commit messages. To address RQ2, we develop a taxonomy that describes how commit messages convey "what" and "why" information (see Sec. 3.3). Finally, to address RQ3 we propose a model that could identify these well-written messages automatically (see Sec. 3.4). Figure 2 presents an overview of this approach. The appendix offers a replication package [71].

3.1 Data Collection and Preprocessing

To ensure that our sample would contain sufficient high-quality commit messages, we selected active and popular projects with a high level of collaboration. We assumed that those projects would have at least some non-trivial portion of good commit messages. Considering the impact of different programming languages on software development [11], in this study, we focused on projects written in Java, one of the most popular programming languages in GitHub [30] and widely used in industry. We selected the top 100 Java projects sorted by their "star" rating in GitHub. While reviewing these projects, we prioritized projects that had previously been subject of studies that focused on automatic generation of commit messages. By addressing some of the limitations of those studies, we seek to offer the results of this study in future improvement of those prior studies focusing on commit message generation. As a result, we selected five OSS projects for this study, listed in Table 1. ***Spring-boot*** [64] helps developers create and run Spring-based applications with less configuration. ***Apache Dubbo*** [9] is a high-performance micro-service development framework. ***Okhttp*** [65] is a HTTP client. ***Junit4*** [70] provides the ability to write repeatable unit tests. ***Retrofit*** [66] is a type-safe HTTP client for Android and Java. Each of the five widely investigated [6, 16, 24, 43, 74, 76] projects has more than 150 contributors, over 8,000 stars, and thousands of commits, indicating that there is active collaboration happening within these projects.

We collected all the commits from the five projects using GitHub's REST API [31] up to February 2021. In the data collection, we only considered commits in which the messages are written in English. This resulted in a dataset containing 41,886 commits (see Table 1). We eliminated commit messages generated automatically by tools (bot messages) based on fixed patterns because this study focuses on messages written manually. Based on the patterns identified by existing work [7, 22, 23, 27], these bot messages can be easily identified and filtered. Table 2 shows several patterns of bot messages in our dataset, which were excluded (ignoring cases). After this data cleaning step, 29,348 commit messages remained.

Table 1: Dataset summary statistics (until Feb 2021)

Project	Start	#Contrib.	#Commits	#Cleaned
Spring-boot	Oct-2012	812	30,072	21,169
Apache Dubbo	Jun-2012	404	2,687	2,249
Okhttp	Jul-2012	236	4,800	2,817
Junit4	Apr-2009	151	2,467	2,035
Retrofit	Sep-2010	152	1,860	1,078
			41,886	29,348

3.2 Identifying Well-Written Messages

It is known that the quality of commit messages varies [26, 45, 47], but a widely recognized standard of high-quality messages is as of yet lacking. Before investigating the characteristics of well-written messages, we constructed the standards to identify them via a survey of both academic papers and developer forums and validate the standards with experienced OSS developers, as described below:

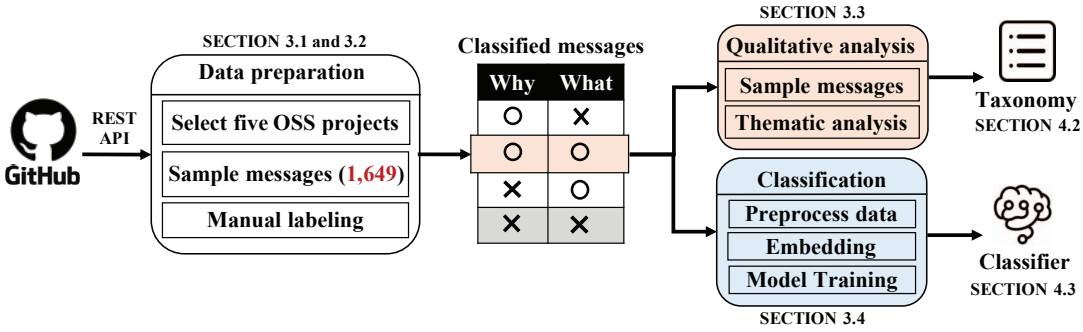

Figure 2: Overview of our approach

Table 2: Non-human written message patterns

No.	Pattern
1	merge branch <branch> (of <project url>) (into <branch>)
2	merge remote-tracking branch <branch> (into <branch>)
3	[maven-release-plugin]
4	...cherry picked from commit <commit url>
5	Next development version <version number>
6	message written by non-human accounts, such as Spring Operator, dependabot[bot], no author

"<branch>" means branch name of the project, "(...)" means optional

- To obtain a scientific perspective of commit messages, we identified and reviewed 46 relevant studies (the full papers are listed in our online appendix [71]), mainly focusing on the expectation of commit messages and whether there is a standard of good messages.
- To obtain a pragmatic and practice-oriented view of commit messages, we used Google's search engine with the phrase "good commit message." In the top 50 results sorted by relevance, we manually selected and studied the online records (and their references) from OSS communities or OSS developers (the whole set of links is included in the appendix [71]). Furthermore, we also solicited opinions from 30 experienced OSS developers to gather their views on what constitutes a good message; we defined "experienced" here as those developers who contributed more than 10 commits to the studied projects.

Through qualitative analysis of the selected records, we observed that the most frequently recognized expectation of a commit message is ***to summarize the changes in this commit (noted as 'What') and describe the reasons for the changes (noted as 'Why').*** The surveyed developers showed a high degree of consistency in the content of commit messages: approximately 93% of them held the view that a commit message should summarize *what* was changed, and describe *why* those changes are needed. That is, rather than only summarizing changes or describing the reasons for the changes, a commit message should have both What and Why information to help collaborators understand the changes.

Therefore, we conducted this research based on the hypothesis that a good commit message should contain a justification (i.e., **"Why"**) that describes the motivation of the change, and a change summary (i.e., **"What"**). Depending on whether or not the two key elements, Why and What, are included, we divided the commit messages in our dataset into four types: *"Why and What"* (containing both), *"No Why"* (only What information, but no Why); *"No What"* (only Why, but no What); and *"Neither Why nor What"*. We note that Why information for certain changes might be common sense. For example, the commit message "`fix typo a->an`" omits the reason of making this commit, which can be easily inferred as "improving readability." From the perspective of reducing developers' workload, we did not classify these commit messages as "No Why" because the rationale is trivial. Similarly, some What information could be easily inferred from `diffs`,[1] and we took the same approach as we did for Why. More details of the categories of common sense Why and easy-to-infer What can be found in Sections 4.2.1 and 4.2.2. Further, we found some commits express Why information by providing a link to an issue report or pull request, which usually includes a detailed motivation and discussions of the change [8, 82]. This approach is controversial, however: one school of thought argues that these linked resources provide a convenient way to offer full details that lay out the rationale for a change [60, 68]. Another school of thought argues that such links pose a risk as they might go stale, resulting in a loss of the information they point to, thus resulting in commit messages that are difficult to understand [41]. In this study, we took the former view and treated links of issue reports and pull requests in commit messages as a way to provide Why information.

To study the distribution of the four message types in the selected projects (Table 1), we used the clustered random sampling technique [5] on the five projects' commit data and selected 1,649 commit messages (confidence level: 95%, margin of error: 5%), which were committed by 339 developers. On average, each developer submitted approximately five commits in our dataset. The first two authors of this study labeled the commit messages independently, categorizing each into one of the four message types. During the labeling process, we identified and eliminated 52 non-atomic commits, where more than one change was submitted. Making multiple

[1] `Diffs` are raw content of changes generated using the *git diff* command to show differences between different versions of commits.

changes in a single commit is considered bad practice as it may reduce maintainability [2, 10, 62]. After labeling the messages (1,649 minus 52 that were removed), Cohen's kappa coefficient of agreement [17] between the two authors was 0.91. As for the messages labeled differently, we held several meetings to resolve 66 (approx. 4.1%) disagreements. If the first two authors failed to reach an agreement on the type, a third author acted as an arbitrator. Moreover, we validated the labeled results by conducting a survey with OSS contributors. Specifically, we selected 958 experienced contributors (i.e., those who contributed more than ten commits) from the top 100 Java projects sorted by their star count and sent them a questionnaire (see the appendix [71]) to solicit their views. We received 30 valid responses (a response rate of 3.1%). After calculating the 120 results labeled by developers (each respondent labeled four commit messages), we found that 102 results were labeled consistently with ours. This indicates that the accuracy of the dataset reached almost 85%.

3.3 Characterizing Well-Written Messages

In the second phase, we sought to identify the characteristics of well-written messages (labeled in Sec. 3.2). We manually sampled 271 (confidence level: 95%, margin of error: 5%) commit messages with a labeled type as *"Why and What"*—that is, messages that contained both Why and What information, and thus did not miss important information. Among the 271 commits, we removed 19 commits that only use links of pull requests or issue reports to express Why. For the remaining 252 messages, we used thematic analysis [20] to characterize how developers express Why and What information in the commit messages, according to the following process. (1) We first read and analyzed all the commit messages, to understand how developers described code changes and motivation, and identified phrases that expressed Why and What. (2) We reread the whole commit messages and related phrases carefully to generate initial codes and organize them in a systematic way. (3) After completing the generation of the initial code, we aggregated codes with similar meanings, and identified an initial theme representing that cluster. After this step, all codes were divided into one of the initial themes, which helped in identifying any emergent patterns that characterized the descriptions of Why and What. (4) We then reviewed the initial set of themes to identify opportunities to merge similar themes. By clarifying the essence of each theme, similar themes were merged into a new theme, or a theme was included as a sub-theme. (5) In the last step, we defined the final set of themes.

To reduce any researcher bias, steps (1) to (4) described above were performed independently by the first two authors [57]. After this, a sequence of meetings was held to resolve conflicts and assign the final themes (step 5).

During thematic analysis of the 252 commit messages, we found the way developers describe Why and What in messages tends to vary across different types of maintenance activities. Therefore, we also classified commit types to investigate the relationship between message expression categories and maintenance activities. Prior literature provides a variety of code change classifications [49, 56, 67, 73], but no consensus was reached regarding the different types of classification to which a commit refers. Therefore, after an ad-hoc literature review, we adopted the widely used definition of Mockus and Votta [49] for three commit types: (1) corrective changes address processing, performance, and implementation failures; (2) adaptive changes represent changes in the data environment or processing environment. For example, to implement a new function; and (3) perfective changes, which focus on improving non-functional attributes such as efficiency, performance, cleanup, etc. Then, we identified commit type by deductive thematic analysis [53], which can match the data with themes from extant research. More specifically, the first two authors independently took the theoretical propositions derived from Mockus and Votta [49] as a point of departure, and applied them to the 252 commit messages. We obtained a high level of consistency (Cohen's kappa coefficient = 0.92) between the two coders. The two coders discussed and resolved any disagreements.

3.4 Automatic Identification

In the third phase, we sought to develop a solution that could automatically identify well-written messages. As we defined high-quality commit messages as those containing both Why and What information, we first designed two classifiers that could automatically identify whether a message contains Why (labeled ***C-Why***, and ***C*** means "classifier") and What (labeled ***C-What***) separately. Training the two separate classifiers can offer more fine-grained feedback to developers by indicating which of the two key elements (Why and What) is missing. We then selected and combined the two classifiers with the best performance to automatically identify well-written messages that contain both Why and What (labeled as ***C-Good***).

3.4.1 Data Preparation. We used the commit messages labeled in Sec. 3.2 to train and test the three classifiers. Commit messages usually include several tokens that are not "natural language," such as links to pull requests. Since their full semantics are highly specific to the contents of the commits, which we do not consider in this paper, we replaced these tokens with placeholders indicating the kind of information, to ensure the models were not affected by such trivial commit content. Specifically, we identified and replaced the following tokens of non-natural language: 1) we replaced any URLs in a message with "`<X url>`", where "X" refers to the types of URLs, i.e., "pr" (indicating pull request links), "issue" (indicating links of issue reports), and "other." 2) We replaced code elements in the messages with "`<X name>`", where "X" refers to the types of code elements, such as method and file. These code elements were identified by comparing messages with the corresponding code changes. 3) We retained the paragraph information for commit messages by replacing newline characters with "`<enter>`".

3.4.2 Identification. It is likely that the original dataset is imbalanced, i.e., the number of messages that contain Why (or What) is larger than the messages that do not contain this information. However, imbalanced datasets can cause machine learning (ML) models to focus on major categories and undervalue other minor categories [3] that we are more concerned about in this paper. ML-based classifiers often use over-sampling methods to solve the data imbalance problem so as to achieve better performance. To this end, we tried three widely used over-sampling techniques, including random sampling with replacement [37], synthetic minority

oversampling technique (SMOTE) [15], and the adaptive synthetic (ADASYN) [33], to prepare the data before training the classifiers, and selected the technique that produced the highest accuracy in each classifier.

Next, the input textual messages were vectorized. The vectorization method, i.e., Bidirectional Encoder Representations from Transformers (BERT) [21], has been shown to exhibit good performance in natural language processing tasks including text classification [1, 29]. We used BERT to embed the tokenized and padded commit messages and convert each message into a numeric vector. A large number of techniques have been proposed to solve automatic classification tasks [48]. We considered the most widely-used techniques to classify commit messages, including Long Short-Term Memory (LSTM) [34], bidirectional Long Short-Term Memory (Bi-LSTM) [61], Multi-Layer Perceptron (MLP) [52], Logistic Regression [48], Random Forest [12], K-Nearest Neighbors (KNN) [19], Gradient Boosting Machine [28], and Decision Tree [55]. We evaluated their performance in our classification task, and selected the best performing approach.

4 RESULTS

We now present the results of our study, addressing the three research questions outlined in Sec. 1. Sec. 4.1 presents the distribution of different types of commit messages. Sec. 4.2 presents a taxonomy of well-written messages, comprised of expression categories that developers use to describe What and Why information. Sec. 4.3 presents the performance results of our automatic classifier of well-written messages. To facilitate traceability, we provide quote messages, and provide identifiers in our database [71].

4.1 Quality Distribution of Commit Messages

We manually classified 1,597 commits (sampled from five OSS projects) into four types based on whether their messages contain Why and What information (see Sec. 3.2). We calculated the distribution of these four types of commit messages in the five OSS projects. Figure 3 presents the results. We can see that the dominating type in four projects (except Retrofit) is well-written messages, i.e., these messages contain both Why and What information. The ratio of this message type in the five projects varies from ca. 42% to ca. 82%, with an average ratio of ca. 56%, suggesting that around 44% of commit messages have quality issues. This, in turn, suggests that any automated approaches to generate commit messages which are trained using datasets containing such large portions may be compromised, as the generated messages may have learned from such incomplete messages. While our sample of five projects is clearly not representative of the larger corpus of Java projects on GitHub, this finding does highlight a potential issue in terms of the effectiveness of existing tools.

As for the three questionable types of messages, "No Why" (containing only What information) accounts for the largest proportion with an average of 28%, and is approximately twice the ratio (12% on average) of the messages containing only Why information. It may indicate that writing the reasons of code changes is more challenging than describing what was changed. Further, the widely different proportions of "No Why" and "No What" may explain why generating Why information for code changes is harder than

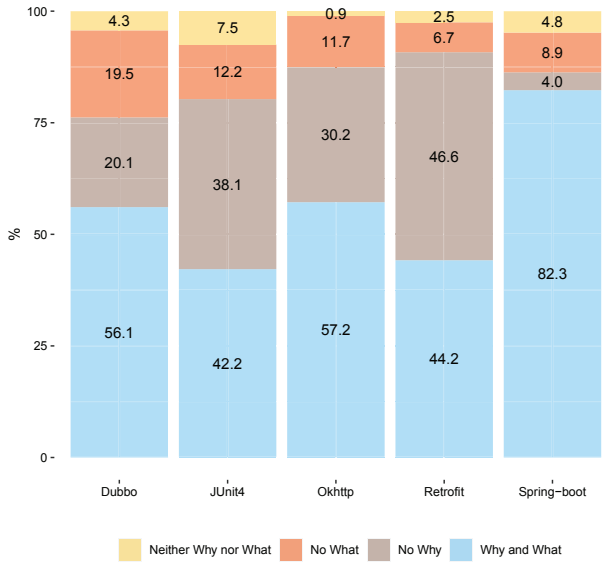

Figure 3: Distribution of the four types of commit messages

generating What information, which previous studies have borne out [44–46].

The type of messages that contain neither Why nor What accounts for the smallest proportion, ranging from ca. 1% to ca. 8%. To further investigate what information the type "Neither Why nor What" exactly contains, we manually analyzed all the 65 messages of this type. Following the same set of steps of thematic analysis [20] described in Sec. 3.3, we identified five categories, each with unique characteristics:

- **Single-word message**: containing only one token which hardly expresses any information. "*Merge*", "*Polish*", and "*<file name>*" are typical examples in our labeled data.
- **Submit-centered message**: expressing nothing but the fact that it is a commit. For example, "*Loader changes.*" It is obvious that this is a commit, but it is impossible to know what the changes are, and why they were made.
- **Scope-centered message**: explaining nothing but the scope of the changes. Typical messages of this category include "*minor changes in test.*" Most of such messages contain qualifying words like "major" and "minor" and cannot express other information.
- **Redundant message**: describing content that is easy to infer from code diffs. For example, "*Add <file name>*", "*Delete <file name>.*" Even without reading this message, the fact that a file was added or removed can be easily established.
- **Irrelevant message**: describing something irrelevant to the change. A message such as *"Kent & Erich patch swallowing in Merlin"* is written to commit a non-empty message, which conveys no information at all.

The analysis above clearly demonstrates that a considerable portion of commit messages is problematic, i.e., they do not contain important information. That is, they either miss What or Why

information, or both. As we discussed in Sec. 2, several message generators have been trained and evaluated using commit messages as datasets, without filtering out sub-optimal messages. Consequently, a major threat was introduced due to using sub-optimal data. The fourth type, "Neither Why nor What", might be easy to identify and remove, but this type only accounts for a limited proportion. In other words, filtering this type of message cannot address the threat sufficiently, and a more powerful automatic identifier of good messages is needed.

Summary for RQ1: The quality of commit messages varies in the five studied OSS projects, with on average ca. 44% of messages in need of improvement. Further, we identified five categories of messages that contain neither "Why" nor "What."

4.2 A Taxonomy of Commit Messages

We now turn to RQ2, which seeks to shed light on what makes a well-written commit message. We identified the various categories of rationales (the "Why") as well as the contents (the "What"), and analyzed the relationships between these categories and the typology of Mockus and Votta [49]. Note that multiple categories of Why (or What) may exist in the same message. Specifically, we introduce these categories in Sec. 4.2.1 and 4.2.2, and present the prevalence of these categories in Sec. 4.2.3. We summarize the detailed codes and statistics of the Why and What categories in our online appendix [71].

4.2.1 Why Expression Categories. By analyzing the various ways in which developers express the change rationale (i.e., Why), we found that developers tend to express this directly or indirectly, or sometimes not at all when the reason for a change is common sense or can be explained by the change itself. Using the procedure outlined in Sec. 3.3, we identified five main categories with 18 subcategories. We introduce these (sub)categories with examples as follows, including their counts and percentages (indicated in parentheses) in the 252 analyzed commit messages.

Describe Issue. This category directly elaborates the motivation of a code change; it is mainly concerned with an issue in the current code implementation. Developers explained their motivation for a change by describing the issues and the specific scenario in which they occurred. This makes the context of a commit easier to understand for other contributors. We identified three subcategories:

- DI1 **Describe error scenario** (#50, 19.8%): this subcategory directly describes where and how an error occurs. It is the most common way of describing Why a change was made. Developers frequently indicated the source of a bug, or specified the steps for reproducing it, and also explained their impact; for example: *"As-is it throws unchecked exceptions on unexpected charsets. This is a problem because it can cause a misbehaving webserver to crash the client."* [#S243]
- DI2 **Introduce issue report** (#11, 4.4%): this subcategory describes issues mainly by citing errors/defects or warnings from quality assurance tools. Some recognized or common tools are usually chosen to achieve the contributor's common understanding of the mistakes. For example, *"remove warnings found by errorprone. [...] CallTest.java:2056: warning: [UnnecessaryParentheses] Unnecessary use of grouping parenthese [...]"* [#S88]. This message cites a warning message from the tool "errorprone" [32] to describe the motivation for this commit.
- DI3 **Describe shortcoming** (#9, 3.6%): this subcategory highlights the shortcomings or weaknesses in the current implementation, which is the motivation to make this commit. For example, *"I'm unhappy with java.io: No timeouts [...] Features like mark/reset and available() are clumsy [...]"* [#S141].

Illustrate Requirement. The second category, *Illustrate Requirement*, describes the source of requirements that led to this commit. These requirements include the need for software development and addressing problems in the process of software maintenance. We identified three subcategories:

- IR1 **Usage need** (#11, 4.4%): these commit messages describe specific needs or requirements of users in software development. This message helps other contributors understand the background and necessity of this change. For example, *"Error-prone only works on pre-12 at the moment and we need this configuration to apply for all JDKs"* [#S225].
- IR2 **Out of date** (#24, 9.5%): these commit messages indicate the obsolescence of some features or code. This includes the deprecation and subsequent removal of unused code objects such as classes, methods, or attributes as the software evolved. Other object version upgrades may also cause a dependency to become obsolete requiring modifications. For example: *"Remove outdated key. The 'spring.metrics.export.redis.aggregate-key-pattern' is no longer defined but was still referenced in the documentation."* [#S127].
- IR3 **Runtime or development environment change** (#11, 4.4%): commit messages in this subcategory indicate an adaptation to the current code development or runtime environment. This includes changes to the implementation to adapt dependent functional changes, modify documents, return values, or examples to accommodate API modifications, etc. For example, *"API has changed, fixing the example,"* indicates that the developer changed the example to match the changes of an API [#S142].

Describe Objective. The third category of rationale provides the purpose of a change, such as the future prevention of defects or optimization of functionality or performance. We identified two types of objectives:

- DO1 **To fix defects** (#9, 3.6%): these commit messages make explicit that the purpose of a change is to resolve a defect. Different from `Describe error scenario` (DI1) that describes how a defect occurs or can be observed, this type of expression clarifies how the proposed change will *resolve* a defect, as in this example: *"Fix concurrent problem of zookeeper configcenter, wait to start until cache being fully populated"* [#S250].
- DO2 **To make improvements** (#13, 5.2%): a message in this subcategory directly describes the improvement of an author's code implementation, e.g., functional improvement or non-functional goals. The message explains the reasons for the

change by describing a specific promotion goal. Commit messages in this category usually use phrases such as *to do something* or *for something* to describe what an author seeks to achieve. For example, the message *"AndroidLog: Added [...] methods for easier subclassing."*[#S181] clarifies that the change will simplify the act of subclassing.

Imply Necessity. Different from commit messages in the previous three categories, messages in the *Imply Necessity* category only *indirectly* describe the need for changes. Developers indirectly described the necessity of changes using the messages that fall into the following subcategories:

- IN1 **Conventions and standards** (#15, 6.0%): this subcategory describes or refers to any conventions or standards that are the basis for a change, thus demonstrating a sound rationale of the change. Conventions are agreements among developers within the project, or could also be industry-wide conventions. Standards are written specifications, often rather technical and more formal, and thus are more rigid than conventions. A common understanding among contributors expressed in commit message makes a commit easier to understand. For example, one developer referred to a convention regarding the location of tests, thus explaining why the commit moves the location of the test: *"it is common to add tests to the same package as the class under test"* [#S8].
- IN2 **Relation to prior commits** (#9, 3.6%): these commit messages explain the relationship between the current commit and any commits that were already merged into the repository. These messages clarify the motivation by improving any problems with a prior accepted change or using the new features introduced by a previous change, etc. Therefore, while the `Describe Issue` category may be used to indicate that there is a problem *before* changing it, the nature of this subcategory is to introduce prior accepted changes as the context for the current commit. For example, this commit adds a test because *"the code was changed by commit <other url> but unfortunately the test was not part of the commit"* [#S40].
- IN3 **Relation to an implemented feature** (#9, 3.6%): some commits are related to an existing feature, and this relationship provides the context of the new commit. The current change is part of a large operation that may be underway, such as the message [#S44] indicates that the current change is *"a short step on the road to HTTP body format agnostic support."* The current change may also be preparation work for an accepted feature, such as message [#S41] which directly explains that the change will *"make a future change easier to land."* In the context of established goals (achieving functionality or larger operations), the motivation for this change will be clear.
- IN4 **Improvements and benefits** (#39, 15.5%): the last subcategory refers to commit messages that indirectly describe the need for a change by explaining the improvements and benefits that the change will bring. Such commits may include either functional or non-functional improvements such as readability and maintainability. The commit message may also include a comparison between "before" and "after" the commit, which gives collaborators a better understanding of the motivation for change. For example, the following commit message suggests a proposed improvement and the associated benefit: *"Use custom exception type [...]. Since we omit the stack trace, this more clearly indicates the source being from Retrofit's mock behavior"*[#S103].

Missing Why. This category includes commit messages that do not offer a rationale, for example, when it is common sense or easy to infer. In such cases, there is no need to provide a rationale. These include the following six subcategories:

- MW1 **Test cases** (#4, 1.6%): these commits involve adding test cases to the repository; in many projects, there is consensus among developers to add test cases for each feature. For example, *"tests for canceling async requests."* [#S21].
- MW2 **Typographic fixes** (#7, 2.8%): these commits involve the correction of typographic errors. Fixing such errors helps to increase the correctness and readability of code or documentation, for example: *"fix typo a->an"* [#S4].
- MW3 **Text file changes** (#13, 5.2%): these commits involve changes made only in text files. Such files have specific functions, such as "ChangeLog" files that record changes, "README" files that outline the project, and so on.
- MW4 **Annotation changes** (#5, 2.0%): annotation changes specifically refer to the motivation to modify the content of annotations. Annotations are descriptions of code objects, and their main purpose is to increase the readability of the code, so the "Why" for comment changes is common sense. For example, *"add docs about null responses"* [#261].
- MW5 **Code refactoring** (#15, 6.0%): these changes involve refactoring and formatting of code. Changes may include polishing, formatting, renaming, cleaning up and other similar operations to improve the readability of the code. For example, *"Polish pom.xml. Apply consistent formatting, drop JDK 8 support and cleanup repo [...]"* [#S47].
- MW6 **Version management** (#5, 2.0%): these commits include changes that involve version management, such as the updating of version numbers. This is an essential step as it tags a specific version of the software, which is important for maintainability, e,g., *"prepare version 2.8.1"* [#S150].

4.2.2 What Expression Categories. We identified four categories to express change, i.e. how to express "What" in commit messages. We introduce these categories with examples as follows, including their counts and percentages in the 252 analyzed commit messages (indicated in parentheses).

Summarize Code Object Change. The first category represents commit messages that summarize the changes; effectively a summary of the `diffs`. We identified the following subcategories:

- SC1 **Characteristics of changes** (#13, 5.2%): this subcategory highlights the characteristics of the current code change and compares them with other alternative implementations to summarize code changes. For example, in this commit message the developer described an *"attempt at a 3rd I/O interface"*, and described the implementation as being *"inspired by InputStream and OutputStream, but using growing buffers*

instead of byte arrays as the core data container" [#S141] (advocating against fixed-size byte arrays).

- sc2 **Object of change** (#143, 56.8%): commit messages in this subcategory summarize the changes from the point of view of the code objects. Over half of the commits express What was changed by pointing out the changed object, which refers to the key component of this change, and developers highlight this in the message. These code objects include attributes, methods, classes, packages, and so on. For example, *"remove creation of 'fat' jar..."* [#S157].
- sc3 **Change list** (#6, 2.4%): commit messages in this subcategory indicate changes of several code objects, involving one or more source files. For example, *"this commit removes the following deprecated properties: * 'server.connection-timeout' * 'server.use-forward-headers' [...]"* [#S154].
- sc4 **Contrast before and after** (#16, 6.4%): messages in this subcategory contrast the state of code objects before and after changes. The following is an example of a contrast before/after message: *"rename HeldCertificate.Builder.issuedBy() to signedBy()"* [#S64].

Describe Implementation Principle. This category represents commit messages describing technical principles underpinning the changes. The implementation rationale shows the process by which the code executes correctly. For example, *"SslContextBuilder was using InetAddress.getByName(null) [...] On Android, null returns IPv6 loopback, which has the name 'ip6-localhost' "* [#S251]. Only six commits (out of 252, 2.4%) fell in this category.

Illustrate Function. Commit messages in this category summarize and explain code changes from a functional perspective. Unlike describing specific changes in code, these messages pay more attention to functional changes. Such messages inform other contributors what has changed by describing any new behaviors introduced by these changes. For example, *"Rename preferred-mapper property so its clear it only applies to JSON"* [#S169]. This category is common, 65 out of the 252 analyzed commits express what was changed by illustrating function.

Missing What. This category refers to commit messages that lack any specification of what was changed. Typically any such changes are small and simple that can be easily inferred. We find this category in 19 commits (accounting for 7.5%). Common examples are the correction of typographic errors, renaming of source code objects, and adding and removing spaces.

4.2.3 Linking Maintenance Dimensions to Commit Messages. As discussed in Sec. 2, the nature of maintenance activities varies by type as defined by Mockus and Votta [49]. Different types of maintenance activities (as per the typology of Mockus and Votta [49]) tend to take different ways to describe changes. We analyzed the distribution of the expression categories of Why and What in the different development activities (see Table 3). It is worth noting that some developers use multiple (but no more than two) expression categories together when describing Why or What, and these messages account for only a small percentage, i.e., no more than 9% of the 252 messages.

Corrective changes are performed to fix defects in an existing codebase. As shown in Table 3, this matches our finding that

Table 3: Expression categories across maintenance activities

	Category	Corrective (#116)	Adaptive (#63)	Perfective (#73)
How to express "Why"	Describe issue	**45.7%**	12.7%	6.9%
	Illustrate requirement	12.1%	22.2%	21.9%
	Describe objective	6.9%	7.9%	11.0%
	Imply necessity	19.0%	**39.7%**	26.0%
	Missing Why	12.1%	15.9%	**34.2%**
	Describe issue & Describe objective	0.8%	0 %	0 %
	Describe issue & Imply necessity	2.6%	0 %	0 %
	Illustrate requirement & Imply necessity	0.8%	1.6%	0 %
	Total	100.0%	100.0%	100.0%
How to express "What"	Summarize code object Change	**58.6%**	**60.3%**	**76.7%**
	Illustrate function	22.4%	27.0%	8.2%
	Describe implementation principle	4.3%	1.6%	0 %
	Missing What	6.1%	3.2%	13.7%
	Summarize code object change & Illustrate function	8.6%	7.9%	1.4%
	Total	100.0%	100.0%	100.0%

`Describe Issue` is the most common expression category to explain the reason for corrective changes, with the highest proportion of 45.7%. Developers also express the rationale of code change by combining `Describe Issue` with `Describe objective` (0.8%) and `Imply necessity` (2.6%). Likewise, `Summary of code object changes` is the most common category to describe the What for this maintenance type in a commit message, at 58.6%.

For Adaptive changes, developers often describe Why by indirectly implying the necessity of the change (category `Imply Necessity`), with the highest proportion of 39.7%, and followed by the `Illustrate Requirement` category, which clarifies the requirements that underpin the change. A possible reason might be that a new feature or change in the processing or data environment usually indicates a need for a change when it is the first commit. However, the implementation of most new features requires multiple changes, and a developer can explain the motivation of a change by describing the relationship with a feature or change what was accepted previously. Another reason may be the description of a change's improvements or benefits to support adding new features.

Perfective changes that developers make to improve, for example, code readability and quality, usually involve only text files, comments, or tests. The motivation for these changes tends to be common sense, so Why information is frequently omitted in the messages (category `Missing Why`), with a proportion of 34.2%. For other non-functional properties, developers tend to use the `Imply necessity` and `Illustrate requirement` categories.

We can see that the most common way of expressing Why varies with the types of maintenance activities, i.e., `Describe Issue` for corrective changes, `Imply necessity` for adaptive changes, and `Missing Why` for perfective changes. However, developers tend to summarize code changes directly when describing changes. In addition to perfective maintenance activities, summarizing the changes from the perspective of functional changes is also a common way for developers. The possible reason is that the improvement of non-functional properties is not reflected in the functionality offered by the software. The `Missing What` category is unusual for all three maintenance activities.

Summary for RQ2: We identified five expression categories of "Why" and four expression categories of "What." Further, we found that developers have different expression preferences when writing commit messages for different activities. The results can help developers write a good commit message.

4.3 Automatically Classifying Good Messages

We performed a ten-fold cross-validation to estimate the classifiers' performance. The 1,597 messages were randomly partitioned into ten subsets of similar size. The validation had ten rounds; in each round, nine subsets were used to train the model, and the remaining one was used for testing. A different subset was used for testing in each round. We reported the average performance of the ten rounds. To investigate the impact of different classification techniques on the performance of our approach, we achieved our classifiers based on eight common classification techniques respectively (see Sec. 3.4) and repeated the evaluation on the same dataset. Table 4 presents the performance of ***C-Why*** and ***C-What*** using different classification techniques. These results indicate that Bi-LSTM has the best performance on both ***C-Why*** and ***C-What***, with an accuracy of 84.7% and 91.0%. This result is consistent with prior findings [40, 80], namely that deep learning based neural networks are better at processing text classification tasks. Therefore, we chose Bi-LSTM to build our classifiers.

Table 4: Ten-fold cross validation of message classification techniques

Techniques	Accuracy C-Why	Accuracy C-What
Bi-LSTM	**84.7%**	**91.0%**
LSTM	83.6%	90.1%
Logistic Regression	79.1%	85.5%
MLP	80.3%	85.1%
Random Forest	76.5%	86.1%
Gradient Boosting	72.5%	77.1%
KNN	74.5%	70.7%
Decision Tree	68.5%	76.8%

Table 5 shows that the three Bi-LSTM based classifiers perform very well, with an accuracy of 84.7%, 91.0%, and 75.9%, respectively. Specifically, when determining whether a message misses "Why" (*Positive*: missing Why, *Negative*: having Why), our classifier ***C-Why*** exhibits good performance with a precision of 76.5% and a recall of 70.9%. In determining whether a message misses "What" (*Positive*: missing What, *Negative*: having What), our classifier ***C-What*** shows good performance with a precision of 78.2% and a recall of 64.5%. According to the output of our classifiers, developers will get a hint of what information is currently missing, so they can review and revise their commit messages. Further, our classifier ***C-Good*** also exhibits good performance when identifying well-written messages (*Positive*: well written, *Negative*: needs improvement), with a precision of 81.6% and a recall of 74.0%. Compared to the unfiltered dataset with an average of only 56% good commit messages, our classifier can automatically identify and construct a higher-quality commit message dataset.

Table 5: Performance of Bi-LSTM based classifier (ten-fold cross-validation)

Metrics		C-Why	C-What	C-Good
Positive	Precision	76.5%	78.2%	81.6%
	Recall	70.9%	64.5%	74.0%
	F1	73.1%	68.9%	77.6%
Negative	Precision	88.1%	93.4%	70.0%
	Recall	90.2%	96.2%	78.4%
	F1	89.1%	94.7%	73.9%
	Accuracy	84.7%	91.0%	75.9%

Summary for RQ3: We proposed three classification models based on *Bi-LSTM* to automatically identify whether a commit message is well-written and whether a commit message contains "Why" or "What." All of them performed well in our dataset and can be reused.

5 IMPLICATIONS

Commit messages are of pivotal importance to facilitate coordination and communication among developers and thus it is important to understand what constitutes good messages and how they are written. We discuss implications for developers and researchers.

Implications for Developers. Our analysis of how the Why and What information is expressed offers developers an understanding of what constitutes a good commit message. At the same time, the results of linking maintenance activities to the message expression categories can prompt developers to write better commit messages. For example, when performing a corrective task, developers could initially choose the most common expression category (applicable in many scenarios), i.e., `Describe Issue`; this would not only improve the commit message, but may also inspire developers to become more aware of different ways to express why changes are needed. More specifically, the subcategory `Describe error scenario` can inspire the developer to describe the bug reproduction steps and the background of the change. At the same time, developers can choose the most popular way, i.e., `Object of change` to summarize code changes and describe the changes to

key code objects so that other developers can grasp the focus more quickly. We hope these can improve the quality of commit messages in the long term.

We also designed two automatic tools for developers to check whether the commit message being written conforms to a good one, i.e., containing both Why and What. With the help of these tools, the developer can know what is missing in his/her commit message and supplement it accordingly.

Implications for Researchers. Our study demonstrates that considerable proportions of commit messages are of poor quality (an average of 44% in the five popular OSS projects), suggesting that this important modality for developers to share knowledge is not used optimally. While this has consequences for the long-term maintainability of any project, we also observed that approaches for automatic generation of commit messages are trained with uncurated data, i.e., datasets with commit messages that are not filtered based on quality. In turn, such models generate low-quality commit messages as well, which exacerbates the issue of poorly written commit messages.

Nevertheless, it is time-consuming for researchers to curate good commit messages manually. We proposed an automatic classifier (i.e., ***C-Good***) that performs well in identifying good commit messages (precision: 81.6%). This classifier has the potential to help researchers construct a large dataset of high-quality commit messages from massive historical data stored in GitHub and other VCSs. In the future, it is necessary to construct a standard dataset of commit messages to facilitate comparison among different generation methods and promote the quality of generated messages.

6 THREATS TO VALIDITY

We are aware of some threats to validity which we discuss next. When constructing the dataset, we removed commit messages generated by known bots [7, 22, 23, 27]. It may be that new bots are becoming more advanced and generating more human-like messages, suggesting a threat where the filtered data may still contain some non-human-written messages. Future work should consider such developments of new bots as they may require more careful data filtering. Notably, among the 1,649 manually analyzed messages, we did not find any new bot patterns, which suggest the results of our qualitative analysis were not affected.

Manual labeling of commit messages poses a subjective threat to validity. To minimize this threat, two authors labeled commit messages independently and introduced an experienced colleague in qualitative research to reach an agreement through several discussions. The agreement level (0.91) is high, indicating a high level of reliability. Some of the labeled messages were subsequently confirmed by experienced OSS contributors. Further, the manually analyzed messages come from 339 developers. It means that multiple commit messages may originate from the same authors, which pose a threat of limiting the diversity of the message taxonomy. One strand of future work can extend this analysis to include more projects (and written in other languages than Java), to verify and, if needed, enrich the taxonomy of commit messages reported in this study.

Another threat is that the dataset (1,597 messages in total) is not large enough for training classifiers. The dataset was randomly sampled from the five OSS projects, and its limited scale is because of the complexity involved in manual labeling and analysis of the commit messages. To reduce this threat, we used a ten-fold cross-validation procedure to get as much valid information (nine-fold) from the dataset as possible and use the average performance on different test datasets to improve the models' capability to generalize [59]. However, developers may write better messages over time. The ten-fold cross-validation, i.e., randomly divided ten subsets, did not consider the influence of time on developers' experience in writing message. This is one potential strand for future work to further add rigor to these results.

When automatically classifying commit messages, other factors may be related to the quality of commit messages. For example, Chahal and Saini [14] proposed 11 format-related metrics to measure the syntax quality of commit messages. Our classifiers only used the text content of messages as input for training. To alleviate this threat, we replaced those format-related elements with a unified token during preprocessing (see Sec. 3.4), such as using "<enter>" to represent line brakes. This preprocessing ensures that syntactic features are considered during the classification of commit messages.

Threats to external validity consider the generalization of our findings. The dataset we analyzed was collected from five popular projects on GitHub implemented in Java, thus posing a threat to external validity. Our findings may not be generalizable to other projects, whether they are open or closed-source, or projects that use other languages. It is very well possible that developers using other programming languages have different message-written patterns that have not been explored in the scope of this work. Future studies could investigate more diverse projects to gain a deeper understanding of what constitutes a good commit message. Notwithstanding these limitations, the automatic classification tools we proposed can be easily adapted to other projects with other languages by simply replacing datasets.

7 CONCLUSION

Commit messages play an important role in collaborating software development and evolution. Nonetheless, the considerable proportions of low-quality messages in OSS projects reflect the difficulties that developers face when writing commit messages, and threaten the effectiveness of existing automatic commit message generation tools. Our study explored the distribution and expression patterns of these well-written messages, linked message expression categories to different maintenance activities, and construct several automatic identification models of good commit messages. Our study findings can help developers write good commit messages and assist researchers to construct high-quality datasets before generating messages automatically.

ACKNOWLEDGMENTS

We are grateful to the software engineers who participated in the survey. This work was sponsored by the National Natural Science Foundation of China (62141209, 61690205, 62172037, and 61772071), and Science Foundation Ireland grants 15/SIRG/3293 and 13/RC/2094-P2.

REFERENCES

[1] Ashutosh Adhikari, Achyudh Ram, Raphael Tang, and Jimmy Lin. 2019. Docbert: Bert for document classification. *arXiv preprint arXiv:1904.08398* (2019).

[2] Kapil Agrawal, Sadika Amreen, and Audris Mockus. 2015. Commit Quality in Five High Performance Computing Projects. In *1st IEEE/ACM International Workshop on Software Engineering for High Performance Computing in Science, SE4HPCS 2015*, Jeffrey C. Carver, Paolo Ciancarini, and Neil P. Chue Hong (Eds.). IEEE Computer Society, 24–29. https://doi.org/10.1109/SE4HPCS.2015.11

[3] Rehan Akbani, Stephen Kwek, and Nathalie Japkowicz. 2004. Applying Support Vector Machines to Imbalanced Datasets. In *Machine Learning: ECML 2004, 15th European Conference on Machine Learning (Lecture Notes in Computer Science, Vol. 3201)*, Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi (Eds.). Springer, 39–50. https://doi.org/10.1007/978-3-540-30115-8_7

[4] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. IEEE, 51–58. https://doi.org/10.1109/IWoR.2019.00017

[5] M. Alvi. 2016. A Manual for Selecting Sampling Techniques in Research. *Mpra Paper* (2016).

[6] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. [n.d.]. A survival analysis of source files modified by new developers. In *International Conference on Product-Focused Software Process Improvement*, Vol. 10611. Springer, 80–88. https://doi.org/10.1007/978-3-319-69926-4_7

[7] Sadika Amreen, Audris Mockus, Russell Zaretzki, Christopher Bogart, and Yuxia Zhang. 2020. ALFAA: Active Learning Fingerprint based Anti-Aliasing for correcting developer identity errors in version control systems. *Empir. Softw. Eng.* 25, 2 (2020), 1136–1167. https://doi.org/10.1007/s10664-019-09786-7

[8] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering*. ACM, 361–370. https://doi.org/10.1145/1134285.1134336

[9] Apache. 2021. Dubbo. https://github.com/apache/dubbo.

[10] CHRIS BEAMS. 2014. How to Write a Git Commit Message. https://chris.beams.io/posts/git-commit/.

[11] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *ACM Trans. Program. Lang. Syst.* 41, 4, Article 21 (Oct. 2019), 24 pages. https://doi.org/10.1145/3340571

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

[13] Rpl Buse and W. R. Weimer. 2010. Automatically documenting program changes. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 33–42. https://doi.org/10.1145/1858996.1859005

[14] Kuljit Kaur Chahal and Munish Saini. 2018. Developer Dynamics and Syntactic Quality of Commit Messages in OSS Projects. In *IFIP International Conference on Open Source Systems*, Vol. 525. Springer, 61–76. https://doi.org/10.1007/978-3-319-92375-8_6

[15] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357. https://doi.org/10.1613/jair.953

[16] Qiuyuan Chen, Han Hu, and Zhaoyi Liu. 2019. Code summarization with abstract syntax tree. In *International Conference on Neural Information Processing*, Vol. 1143. Springer, 652–660. https://doi.org/10.1007/978-3-030-36802-9_69

[17] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[18] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 275–284. https://doi.org/10.1109/SCAM.2014.14

[19] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.

[20] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 275–284. https://doi.org/10.1109/ESEM.2011.36

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. (2019), 4171–4186. https://doi.org/10.18653/v1/n19-1423

[22] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. 2020. Detecting and characterizing bots that commit code. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, 209–219. https://doi.org/10.1145/3379597.3387478

[23] Tapajit Dey, Bogdan Vasilescu, and Audris Mockus. 2020. An exploratory study of bot commits. In *ICSE '20: 42nd International Conference on Software Engineering*. ACM, 61–65. https://doi.org/10.1145/3387940.3391502

[24] Zhang Di, Bing Li, Zengyang Li, and Peng Liang. 2018. A preliminary investigation of self-admitted refactorings in open source software (S). In *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 165–164. https://doi.org/10.18293/SEKE2018-081

[25] Geanderson E dos Santos and Eduardo Figueiredo. 2020. Commit Classification using Natural Language Processing: Experiments over Labeled Datasets. In *ClbSE*. 110–123.

[26] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Computer Society, 422–431. https://doi.org/10.1109/ICSE.2013.6606588

[27] Linda Erlenhov, Francisco Gomes de Oliveira Neto, Riccardo Scandariato, and Philipp Leitner. 2019. Current and future bots in software development. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE / ACM, 7–11. https://doi.org/10.1109/BotSE.2019.00009

[28] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[29] Jianfeng Gao, Michel Galley, and Lihong Li. 2018. Neural approaches to conversational ai. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 1371–1374. https://doi.org/10.1145/3209978.3210183

[30] Github. 2020. The State of the Octoverse. https://octoverse.github.com/.

[31] GitHub. 2021. GitHub REST API. https://docs.github.com/en/rest.

[32] Google. 2021. Error Prone. https://github.com/google/error-prone.

[33] Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. 2008. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence)*. IEEE, 1322–1328.

[34] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[35] Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiangping Chen, Zibin Zheng, and Mingdong Tang. 2020. Learning Human-Written Commit Messages to Document Code Changes. *J. Comput. Sci. Technol.* 35, 6 (2020), 1258–1277. https://doi.org/10.1007/s11390-020-0496-0

[36] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. 2017. Mining Version Control System for Automatically Generating Commit Comment. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 414–423.

[37] Nathalie Japkowicz et al. 2000. Learning from imbalanced data sets: a comparison of various strategies. In *AAAI workshop on learning from imbalanced data sets*, Vol. 68. AAAI Press Menlo Park, CA, 10–15.

[38] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 135–146. https://doi.org/10.1109/ASE.2017.8115626

[39] Siyuan Jiang and Collin McMillan. 2017. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 320–323.

[40] Cannannore Nidhi Kamath, Syed Saqib Bukhari, and Andreas Dengel. 2018. Comparative study between traditional machine learning and deep learning approaches for text classification. In *Proceedings of the ACM Symposium on Document Engineering 2018*. 1–11.

[41] Tien-Duy B Le, Mario Linares-Vásquez, David Lo, and Denys Poshyvanyk. 2015. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 36–47.

[42] Stanislav Levin and Amiram Yehudai. 2017. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 97–106.

[43] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 709–712.

[44] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2019. ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking. *CoRR* abs/1912.02972 (2019). arXiv:1912.02972 http://arxiv.org/abs/1912.02972

[45] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 373–384. https://doi.org/10.1145/3238147.3238190

[46] Pablo Loyola, Edison Marrese-Taylor, Jorge A. Balazs, Yutaka Matsuo, and Fumiko Satoh. 2018. Content Aware Source Code Change Description Generation. In *Proceedings of the 11th International Conference on Natural Language Generation, Tilburg University, The Netherlands, November 5-8, 2018*, Emiel Krahmer, Albert Gatt, and Martijn Goudbeek (Eds.). Association for Computational Linguistics,

119–128. https://doi.org/10.18653/v1/w18-6513
[47] W. Maalej and H. Happel. 2010. Can development work describe itself?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 191–200.
[48] Scott Menard. 2002. *Applied logistic regression analysis*. Vol. 106. Sage.
[49] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes using Historic Databases. In *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*. IEEE Computer Society, 120–130. https://doi.org/10.1109/ICSM.2000.883028
[50] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2021. CoreGen: Contextualized Code Representation Learning for Commit Message Generation. *Neurocomputing* (2021).
[51] Hoorvash Nikoo. 2021. Writing Meaningful Commit Messages. https://dev.to/yvonnickfrin/a-guide-on-commit-messages-d8n.
[52] Sankar K Pal and Sushmita Mitra. 1992. Multilayer perceptron, fuzzy sets, classifiaction. (1992).
[53] Noel Pearse. 2019. An illustration of deductive analysis in qualitative research. In *18th European Conference on Research Methodology for Business and Management Studies*. 264.
[54] Ranjith Purushothaman and Dewayne E Perry. 2005. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 511–526.
[55] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.
[56] Christophe Rezk, Yasutaka Kamei, and Shane Mcintosh. 2021. The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects. *IEEE Transactions on Software Engineering* (2021).
[57] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131.
[58] Eddie Antonio Santos and Abram Hindle. 2016. Judging a commit by its cover. In *Proceedings of the 13th International Workshop on Mining Software Repositories-MSR*, Vol. 16. 504–507.
[59] Miriam Seoane Santos, Jastin Pompeu Soares, Pedro Henriques Abreu, Hélder Araújo, and João A. M. Santos. 2018. Cross-Validation for Imbalanced Datasets: Avoiding Overoptimistic and Overfitting Approaches [Research Frontier]. *IEEE Comput. Intell. Mag.* 13, 4 (2018), 59–76. https://doi.org/10.1109/MCI.2018.2866730
[60] Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. 2018. Why are features deprecated? an investigation into the motivation behind deprecation. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 13–24.
[61] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.
[62] Ben Straub Scott Chancon. 2014. Pro Git. https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project.
[63] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. 2016. On automatic summarization of what and why information in source code changes. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 103–112.
[64] Spring. 2021. Spring Boot. https://github.com/spring-projects/spring-boot.
[65] Square. 2021. Okhttp. https://github.com/square/okhttp.
[66] Square. 2021. Retrofit. https://github.com/square/retrofit.
[67] E Burton Swanson. 1976. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*. 492–497.
[68] Xin Tan and Minghui Zhou. 2019. How to Communicate when Submitting Patches: An Empirical Study of the Linux Kernel. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–26.
[69] Y. Tao, Y. Dang, X. Tao, D Zhang, and S. Kim. 2012. How do software engineers understand code changes? - An exploratory study in industry. In *Acm Sigsoft International Symposium on the Foundations of Software Engineering*.
[70] Junit team. 2021. Junit4. https://github.com/junit-team/junit4.
[71] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2021. What Makes a Good Commit Message? https://doi.org/10.5281/zenodo.5763753
[72] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit Message Generation for Source Code Changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 3975–3981. https://doi.org/10.24963/ijcai.2019/552
[73] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software* 113 (2016), 296–308.
[74] Meng Yan, Xin Xia, David Lo, Ahmed E Hassan, and Shanping Li. 2019. Characterizing and identifying reverted commits. *Empirical Software Engineering* 24, 4 (2019), 2171–2208.
[75] Alexey Zagalsky, Joseph Feliciano, Margaret-Anne Storey, Yiyun Zhao, and Weiliang Wang. 2015. The emergence of github as a collaborative platform for education. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. 1906–1917.
[76] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 334–344.
[77] Yuxia Zhang, Xin Tan, Minghui Zhou, and Zhi Jin. 2018. Companies' Domination in FLOSS Development – An Empirical Study of OpenStack. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg*. IEEE.
[78] Y. Zhang, M. Zhou, A. Mockus, and Z. Jin. 2019. Companies' Participation in OSS Development - An Empirical Study of OpenStack. *IEEE Transactions on Software Engineering* (2019), 1–1.
[79] Yuxia Zhang, Minghui Zhou, Klaas-Jan Stol, Jianyu Wu, and Zhi Jin. 2020. How Do Companies Collaborate in Open Source Ecosystems? An Empirical Study of OpenStack *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1196–1208. https://doi.org/10.1145/3377811.3380376
[80] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau. 2015. A C-LSTM neural network for text classification. *arXiv preprint arXiv:1511.08630* (2015).
[81] Minghui Zhou, Audris Mockus, Xiujuan Ma, Lu Zhang, and Hong Mei. 2016. Inflow and retention in oss communities with commercial involvement: A case study of three hybrid projects. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 13.
[82] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2016. Effectiveness of code contribution: From patch-based to pull-request-based tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 871–882.
[83] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.