A Full-fledged Commit Message Quality Checker Based on Machine Learning

David Faragó
Innoopract GmbH & QPR Technologies
Karlsruhe, Germany
farago@qpr-technologies.de

Michael Färber
Karlsruhe Institute of Technology
Karlsruhe, Germany
michael.faerber@kit.edu

Christian Petrov
Innoopract GmbH
Karlsruhe, Germany
cpetrov@innoopract.com

***Abstract*—Commit messages (CMs) are an essential part of version control. By providing important context in regard to what has changed and why, they strongly support software maintenance and evolution. But writing good CMs is difficult and often neglected by developers. So far, there is no tool suitable for practice that automatically assesses how well a CM is written, including its meaning and context. Since this task is challenging, we ask the research question: how well can the CM quality, including semantics and context, be measured with machine learning methods? By considering all rules from the most popular CM quality guideline, creating datasets for those rules, and training and evaluating state-of-the-art machine learning models to check those rules, we can answer the research question with: sufficiently well for practice, with the lowest F_1 score of 82.9%, for the most challenging task. We develop a full-fledged open-source framework that checks all these CM quality rules. It is useful for research, e.g., automatic CM generation, but most importantly for software practitioners to raise the quality of CMs and thus the maintainability and evolution speed of their software.**

***Index Terms*—commit message, maintenance, quality, text classification, machine learning**

I. Introduction

Motivation. Although code should best be self-explanatory, it is unable to contain all context, such as the implementation decisions (e.g., technical trade-offs) or the reasons for the code change (e.g., the business requirement or bug report motivating the commit). We define context as all relevant information that the code change does not convey by itself. CMs of code repositories that document this context are the most important way to understand the code change [37] and therefore help future development, evolution, and maintenance.

A comprehensive yet easy to understand CM history is one of the most powerful maintenance approaches [26] and strongly benefits the code comprehension and team communication, especially with today's increase in remote work [44]. Despite these advantages, CMs are cultivated by only few software developers (see Sec. VI and [38], [45]) – or in Linus Torvalds' words: "GitHub is a total ghetto of crap commit messages". This is confirmed by our experiment with 5,000 random CMs from GitHub: our framework (see Sec. III) assesses 90% of the CMs as low quality, and 55% as low quality due to missing context. In contrast, for the

[1]Abstraction level F: formatting, SY: syntax, SE: semantics

Table I
Chris Beams' CM quality guideline [26]: rules and examples

Rule	Description	L[1]
R1	Separate subject from body with a blank line	F
R2	Limit the subject line to 50 characters	F
R3	Capitalize the subject line	F
R4	Do not end the subject line with a period	F
R5	Use the imperative mood in the subject line	SY
R6	Wrap the body at 72 characters	F
R7	Use the body to explain what and why vs. how	SE

R5 violated ("fix" used as a noun, not as verb)

```
Linter error fix
```

R7 violated (describes "how", but not "what" and "why")

```
Duplicate zval before add_next_index_zval
```

R7 violated (unclear what was wrong before the change[a])

```
Fix Sass + CSS Modules (#3186)
```

R5, R7 satisfied (simple change, context sufficient for R7)

```
Fix linter errors
```

R5, R7 satisfied

```
Fix running ALTER TABLE statements in Execute SQL tab

This fixes a bug introduced in 73efa11. Because SQLite
reports ALTER TABLE statements to return one column
worth of data, DB4S assumed they are close to a SELECT
statement and therefore did not fully execute them.

See issues #2563 and #2622.
```

[a]"No module-specific actions (like compiling the classnames) got done." [20]

well-maintained [15] CM history of the Linux Kernel, our framework assesses only 3% of the CMs as low quality due to missing context. Applied broadly, e.g., as automatic CM quality reviewing tool run locally or as part of a CI pipeline in the cloud, a CM quality checker could train developers and lead to CM histories with much higher quality, resulting in more maintainable and faster evolving software.

CM quality guideline. Developers that cultivate CMs with high quality often follow certain rules. Table I gives a summary and examples for [26], the most prominent (see Sec. II) CM quality guideline. It sums up most ideas from older guidelines into a 7-rule convention, establishing etiquette for formatting, syntax, semantics, and contextual information: The formatting rules R1 to R4, R6 help readability of CMs and are trivial to verify in an automated way. The remaining two rules

require capable natural language understanding (e.g., for the examples in Table I): Syntactic rule R5 enforces imperative verb mood, which improves consistency since it is used in CMs generated by git, too. R5 also improves understandability because the mood distinguishes descriptions about what the change does from descriptions about the context and the way things worked before the change. E.g., for the CM "Incorrect changes were stored", it is not clear whether storing incorrect changes is the fix or the undesired behavior before the fix. Semantic rule R7 says that a CM should leave out details about "how" a change has been made since code should be self-explanatory, and rather focus on "what" has changed (i.e. summary for understanding) and "why" the change has been made in the first place (reasons for the code change), which is particularly useful for software maintainability and evolution. Hence, R7 demands that the CM provides the relevant context, and thus requires investigating the CM's meaning. The amount of required context depends on the situation: Beams argues that "a single line is fine, especially when the change is so simple that no further context is necessary". One-line CMs are very common, making up 35 % of our samples.

Contributions. So far, there is no practical approach to check CMs on a semantic level (see Sec. II). This article fills this gap and presents a practical and programming language-agnostic framework (see Sec. III) integrating state-of-the-art NLP methods to rigorously check all of Beams' guideline. We determined Beams' guideline as gold standard by surveying research articles and CM guidelines (see Sec. II).

We (two authors and further three experienced software developers) create four labeled datasets (see Sec. IV) to train and evaluate state-of-the-art machine learning models for our framework (see Sec. V): our models achieve state-of-the-art performance in assessing the CM quality on the levels format, syntax (F_1 score of 97.8%), and semantics (F_1 score of 82.9%).

So our contributions are datasets to train machine learning (ML) models, evaluations of ML models, and a framework based on ML to automatically check CM quality according to Beams' guideline. We also implemented a tool based on our framework, enabling everyone to assess commit message quality, both locally and using GitHub workflows. Our contributions are open-sourced[2].

II. Related Work

CM quality assessment. Table II summarizes related work on assessing CM quality. It contains a lot of gray literature: software guides, style guides, blog articles, wiki pages, guidelines with criteria by which to assess the CM quality – there is no formal standard on CM quality.

We pick the guideline [26] by software practitioner Chris Beams for our work because of three reasons: (1) It is the most popular: It is cited in about 10% of the repositories having contribution rules (a `CONTRIBUTING.md` file). Some of the contribution rules do not cite [26], but require good CM semantics more vaguely, e.g., that the CM be "understandable", "meaningful", or "well documented". None of the repositories with contribution rules contradict the semantic rule R7. The only other cited CM quality guideline is [47], in about 6% of those repositories. (2) It is the most complete: it covers all CM aspects mentioned by other guidelines, besides minor variations in formatting rules. (3) It is cited by all five research articles about CM quality assessment.

Table II
CM quality assessment: related work

Year	Source	Type	Level[1]	AE[3]
2022	[55]	Conference article	F, SY, SE	✓
2021	[22]	Software guide	F	✓
2020	[19]	Style guide	F, SY, SE	
2020	[52]	Master thesis	F, SY, SE	
2019	[25]	Master thesis	F, SY	✓
2019	[12]	Tool website	F, SY	✓
2019	[18]	Tool website	F, SY	✓
2018	[29]	Journal article	F, SY, SE	
2017	[15]	Software guide	F, SY, SE	
2017	[13]	Style guide	SY, SE	
2016	[9]	Tool website	F, SY	✓
2016	[10]	Style guide	F, SE	
2015	[24]	Workshop article	-	
2014	[26]	Blog article	F, SY, SE	
2014	[28]	Book	F, SY, SE	
2013	[6]	Blog article	F, SE	
2013	[7]	Wiki page	F, SY, SE	
2012	[5]	Wiki page	F, SY, SE	
2011	[4]	Software guide	F, SY, SE	
2009	[2]	Blog article	F, SY, SE	
2008	[47]	Blog article	F, SY, SE	
2000	[43]	Journal article	-	

Insight 1: Chris Beams' article is the most comprehensive guideline on commit message quality. In our study, we evaluate commit message quality based on its rules.

There is only little research (five articles) in the related work. Only the recently published Tian et al. [55] is closely related to our work: They analyze the reason for a CM to be "good", develop a corresponding taxonomy about how CMs convey "what" and "why" information, and train machine learning models to automatically classify CMs as good. Their best model has a high performance: 77.6% positive class F_1, 73.9% negative class F_1. However, they focus on a thorough theoretic analysis about the taxonomy of commit messages instead of realizing a checker for a CM's usefulness in practice, e.g. according to a guideline like [26]. They (2 experts) only investigate a low number (5) of projects with little variance: all are high quality open-source Java projects about web communication. Furthermore, for a large portion of CMs, they rely on unsuitable pattern matching:

- Any CM with automatically generated parts is excluded, e.g., CMs containing `pull request #` (15% to 30% of all CMs). This is unsuitable since CMs containing

[2]See https://doi.org/10.6084/m9.figshare.22096736 and https://github.com/commit-message-collective/beams-commit-message-checker

[3]Automated evaluation: evaluates criteria in an automated way.

automatically generated parts can also be of low quality and should thus be improved, while others do contain (automatically generated or manually added) text making them high quality CMs.
- The "why" criterion is already fulfilled if the CM contains an issue- or PR-link (about 40% of all CMs), or the word "fix" (about 30% of all CMs). Relevant information hidden behind issue- or PR-links is unsuitable, as described in Sec. VI-B. Pattern matching the word "fix" is unsuitable as this is insufficient to check what R7 demands: that the CM makes clear the reasons why the change was made, clarifying the way things worked before the change and what was wrong with that (see Table I).

In contrast, we (5 experts) investigate 427 projects with high variance (multiple domains, programming languages, cultures), also consider CMs with automatically generated parts, ignore issue- and PR-links, and semantically check whether a CM really conveys what and why vs how.

Insight 2: The only study on the most challenging task of assessing semantic commit message quality (R7 of Beams' guideline) falls short in fully capturing semantics and uses a dataset of limited diversity. We account for those shortcomings.

The other four research articles are still related, but less: Chahal et al. [29] conduct a multi-vocal literature review including gray literature. They determine 11 criteria to measure CM quality and rank the importance of the chosen metrics through an expert survey. These metrics are well covered by [26]: the three most important metrics are variations of R5 and R7, and the atomicity rule that a commit should have one logical change. But atomicity is an attribute of the commit, not its message. Schweizer [52] investigates the fluctuations in quality metrics in commit histories. They use [26] as metric for the CM quality. Mockus et al. [43] analyze commits from a quality perspective by considering commit metadata like number of unique CMs and size of commit comments (smaller messages indicating immaturity). Agrawal et al. [24] study literature to operationalize CM quality metrics. Their Google search for "good commit logs" yields more than 57 million search results. They use the top five hits, which are all gray literature: besides [5]–[7], [47], [26] is one of them.

Automatic CM quality assessment. We found six sources that deal with the automated evaluation of CM quality: The only research paper is [55], which contains a model for automation, but is of limited practical use (see above). Alberto et al. [25] propose a tool to validate a syntactic interpretation of Chris Beams' rules. The other four sources are tools: The open-source tool GitCommitBear [9] automatically analyzes R5 by a syntactic analysis based on classical natural language processing, which faces performance issues. The other three tools try to cover all of [26] but fail on R7: The open-source medical image processing (ITK) software guide [22] checks variations of Beams' formatting rule R1, R2, R6. The open-source project [18] validates all syntactic rules of Chris Beams and heuristically checks R5 using a keyword-based approach (for R7, the script performs no check and only states "Not enforceable"). The open-source project [12] validates all of Chris Beams' rules but R7. For R5 they employ a similar technique to [9] and face similar issues. They state that R7 is too subjective, but we show a substantial agreement between experts assessing R7 in Section IV.

Automatic generation of CMs. Most research about CMs is dealing with their automatic generation out of code changes, without considering the intent behind the code changes [37]: An automatically generated CM is evaluated using Natural Language Generation metrics such as BLEU, which measure its closeness to CMs that humans generated out of the code changes [45], [46]. Thus, this field of work is not helpful for us, but it could strongly benefit from an automatic assessment of CM quality: on [45]'s ground truth of 2,521 CMs (obviously poorly-written already filtered out), our framework assessed 85% to have low quality, 48% due to missing context.

III. Approach

We present a framework that performs the first full-fledged assessment of CM quality, including semantics and context, following Chris Beams' guidelines (see [26] and Table I).

A. The Classification Pipeline

As Fig. 1 shows, the framework has a pipeline architecture to successively perform five tasks, all of which take a CM as input and perform a classification. The order of the tasks is from simplest (formatting) to hardest (semantic), thus giving feedback as fast as possible. With many low-quality CMs already filtered out, harder tasks can focus on CMs with potentially high quality. Classification for the last task yields a rating between 1 (worse) and 4 (best). All other tasks perform binary classifications. A classification checks either a rule of [26] or whether project-specific conventions should be followed instead of R7. If a rule violation is detected, the pipeline outputs a warning and exits (Tasks 1, 2 and 5). If a project-specific convention is detected, the pipeline outputs a corresponding recommendation and exits (Tasks 3 and 4). A recommendation is also issued when the CM has minor deficits in regards to R7 (Task 5). So the pipeline output is either a rule violation warning, a recommendation to follow project-specific conventions or improve R7, or `R1-7 satisfied` for a successful pass through all classifiers, indicating high CM quality according to [26]. For warnings and recommendations issued by the pipeline, developers should review Beams' guideline [26], follow project-specific conventions, seek mentorship or feedback from other members of the development team or review past CMs of the project. Therefore, the output of the pipeline directly contributes to improvement of CM quality.

Overall, our framework addresses the following tasks:

Task 1 (check formatting rules). This checks whether a CM fulfills rules R1 - R4 and R6 (see Table I). It is a minor task, so we do not go into detail.

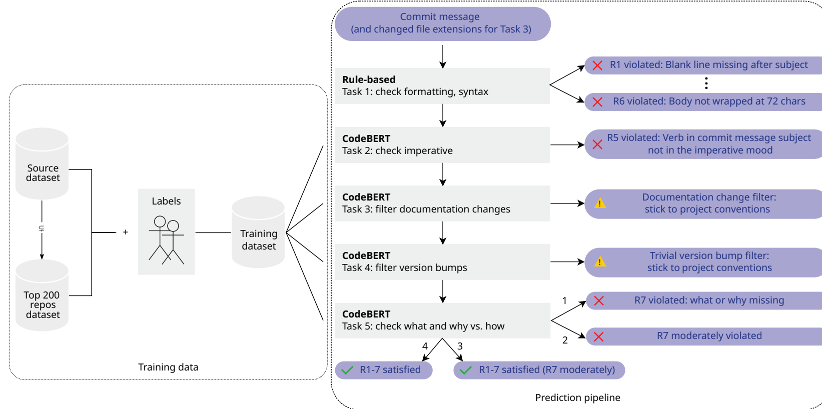

Figure 1. Training data and prediction pipeline for our novel framework for automatic CM quality assessment. It reads commit messages (and a file extension list) and outputs a rule violation warning, recommendation for project-specific conventions or for R7 improvements, or that overall CM quality is good.

Task 2 (check the syntactic rule). Task 2 checks the only syntactic rule, R5: the subject line uses the imperative verb mood. It is challenging due to syntactic ambiguities (see Table I) and the unique vocabulary of CMs [51].

Task 3 (detect documentation changes) & Task 4 (detect trivial version bump changes). Though not part of Chris Beams' rules, these classifiers check whether the CM describes

- a documentation change, i.e., a change that does not modify functional code, only documentation like manuals and code comments (Task 3)
- a trivial version bump, i.e., a change that merely bumps the version of the code project or one of its dependencies, but has no major consequences nor goals beyond the migration to the new version (Task 4)

These rules are not part Chris Beams' guideline, but related to it: The labeling and discussions among the 5 experts revealed that in these two cases (documentation and trivial version bumps), changes are self-explanatory and done routinely. The CMs for these changes need no elaborate explanation and often follow project-specific conventions. Thus, we recommend following project-specific conventions when we detect a documentation change or a trivial version bump change.

Task 5 (check the semantic rule). Task 5 checks the only semantic rule, R7: the body is used to explain what and why vs. how. Task 5 is the main focus of this work and the most challenging task as it requires comprehension of both the CM and further context to decide which context is relevant and should be part of the CM. Thus, Task 5 interprets R7 strictly: It checks not only that the CM text focuses on "what and why" instead of "how", but also that it supplies sufficient "what and why", i.e. all context relevant to fully understand "what and why". R7 originally applies to the CM body. We extend it to the CM summary as well in correspondence to [15] and Beams' argumentation that a single line CM can be sufficient. Since CMs are nuanced with regard to R7, this task does not classify binarily. Instead, a 4-point scale from 1 (violated) over 2 (moderately violated) and 3 (moderately satisfied) to 4 (satisfied) is used, which minimizes central tendency bias (details in Sec. IV-B). Being the last task in the pipeline, its classification result for R7 is output to the user, scores 1 and 2 being warnings that R7 is violated, score 3 meaning that all rules including R7 are met, but recommending to further improve R7, and score 4 indicating high overall commit message quality.

B. Architecture of the classification pipeline

Task 1 is simple, requires no ML, and is implemented as a rule-based classifier. Thus it is easily configurable (e.g., allowing 75 characters in the subject line [15] instead of 50).

For **Task 2-5**, we consider several state-of-the-art machine learning methods. Our final framework uses a language model (LM) based on CodeBERT (see [32]) that we fine-tuned to the corresponding task. In our evaluation in Section V, CodeBERT was for each task either best or among the top 3 best performing classifiers and only marginally worse than the best one (see Table V). Always using CodeBERT simplifies future multi-task learning, common pretraining, and multimodal architectures (see [34]).

For each task, the CM text is passed to the LM's pretrained tokenizer to create input embeddings E. The embeddings are input to the encoder, which produces a contextual representation T summarizing the whole CM. Pretraining with documentation and code enables T to capture contextual knowledge. Using a linear layer, T is projected to the predicted label for Tasks 2-4, and for Task 5 to the interval $[0, 1]$. These are finally mapped to `R7 violated`, `R7 moderately violated`, `R1-7 satisfied (R7 moderately)`, and `R1-7 satisfied` by our pipeline.

Contribution 1: We propose a practical and programming language-agnostic framework for CM quality evaluation based on all rules of Beams' guideline.

IV. Dataset Creation

To the best of our knowledge, there is no suitable labeled data available to train and evaluate our classification Tasks 2-5. Thus, we – two authors and further three software developers, all with 8+ years of industry experience across various domains, programming languages and technologies – create suitable datasets.

A. Dataset Sampling

As depicted in Fig. 1, we use the following data pipeline:

1) two base datasets are created: a diverse *source dataset* based on 1,700 most popular repositories from GitHub and *top 200 repo dataset* $\subsetneq$ *source dataset* with the top 200 of those 1,700 repositories according to their quality.
2) the datasets to train and evaluate Task 2-5 are created out of the base datasets by evenly sampling the data out of the *source dataset* and the *top 200 repo dataset*.

This data pipeline helps to focus on data for labeling and training models that are accurate and robust in various application areas. All datasets are open-sourced[2].

*1) **Source dataset**:* We apply a sampling approach similar to Sarwar et al. [51] and Zafar et al. [57] to ensure a wide variety of software repositories and CM styles, e.g. no restrictions, with CM quality guidelines, Conventional Commits [13], or CM templates. Specifically, we choose the 17 most popular languages on GitHub according to the number of repositories using them. For each of the languages, we choose the most popular 100 repositories according to their number of stars and from each of those 1,700 repositories, we collect their 100 latest commits as those are contained in the data that the GitHub API returns[4].

*2) **Top 200 repo dataset**:* Our *source dataset* is highly skewed towards low quality CMs (see Sec. I). Our framework should help newcomers learn to write good commit messages, as well as teams that already take pride in writing high quality commit messages to further optimize them. To be able to train models that predict accurately for both kind of users, we create the *top 200 repo dataset* which only contains the top 200 repositories, according to our own evaluations.

When creating all datasets to train and evaluate Task 2-5, we evenly sample from the *source dataset* (marked with the label *random pool*) and from the *top 200 repo dataset* (marked with *good pool*), avoiding any duplicates[4].

For Tasks 2-4, we end up with datasets of over 1,250 samples each, covering 644, 564 and 464 repositories respectively. For Task 5, we end up with a dataset of 808 samples, covering 427 repositories.

B. Dataset Labeling

*1) **Task 2-4**:* Our labeled datasets for Task 2-4, annotated according to our annotation guide[2], contain over 1,250 samples with dichotomous annotations each, labeled by 2 experts.

[4]The balancing in our data pipeline reduces labeling effort for reaching a training set variance that is required for the model to generalize sufficiently. The balancing does not hurt validation: the predictive performance (MCC) of our best model for Task 5 only deviates by 2.54% (STD by 2.75%) between our and the original distribution.

*2) **Task 5**:* Since CMs are nuanced with regard to R7, we label on a 4-point scale from 1 (violated) over 2 (moderately violated) and 3 (moderately satisfied) to 4 (satisfied). Compared to the 5-point Likert scale [40], this encourages decisive evaluations and minimizes central tendency bias. To handle uncertain cases, experts could mark commits they were uncertain about, ensuring that we included a labeled commit message only if the corresponding expert was confident. This approach combines the benefits of a forced-choice scale with the ability to handle uncertain cases, resulting in a more reliable and informative labeled dataset. Annotating a dataset for Task 5 is very challenging due to the required implicit knowledge in software development and ambiguities of the context. Thus, we applied the methodology depicted in Fig. 2:

Groupwise. To ensure that our experts are all properly trained and agree on the task, we firstly conducted a feasibility study in form of a groupwise interrater agreement: all 5 experts rated the same set of commits and conducted thorough discussions. We estimated the amount of commits necessary for statistical significance by using the one-sample proportion in the Z-interval. We used $p = 0.5$, a confidence interval of 95% and an error margin of 0.05, resulting in a minimum of 385 samples – we end up using 404 commits. Through the discussions, our agreement rose from $\kappa = 0.42$ to $\kappa = 0.72$, i.e. from "poor" to "substantial" according to Emam et al. [31]. As a result, we created our final annotation guide[2] in form of Chris Beams' article [26] extended by examples and comments.

Pairwise. Based on the final annotation guide[2], experts continued to label a dataset without access to others' ratings. To ensure the highest possible training dataset quality within what is economically feasible, we annotated further data by splitting commits equally among $\binom{5}{2} = 10$ expert pairs. Discussions to resolve disagreements were no longer conducted.

Final dataset. For our final dataset, we filtered out commits with disagreements. Experts marked commits they were uncertain about, which were 18% of the commits they rated, illustrating the difficulty of this task. This leads to 157 commits from the final phase of the groupwise interrater agreement, which have been labeled according to our final annotation guide[2], and 651 commits from the pairwise interrater agreement. In total, the final dataset for Task 5 consists of 808 CMs, 407 stem from the *good pool* and 401 from the *random pool*. For the pairwise interrater agreement, we have an average $\kappa = 0.63$ (substantial) for the *good pool*, $\kappa = 0.60$ (moderate) for the *random pool*, and $\kappa = 0.63$ (substantial) overall. Thus, Task 5 is a feasible task and our final dataset is useful for training and evaluation.

Contribution 2: We create the first labeled datasets for Beams' rules, including the first one to fully capture the semantics of R7. Representing over 400 repositories each, they are diverse. We ensure high-quality labels by involving 5 industry experts.

Disagreement discussion for Task 5. During annotations,

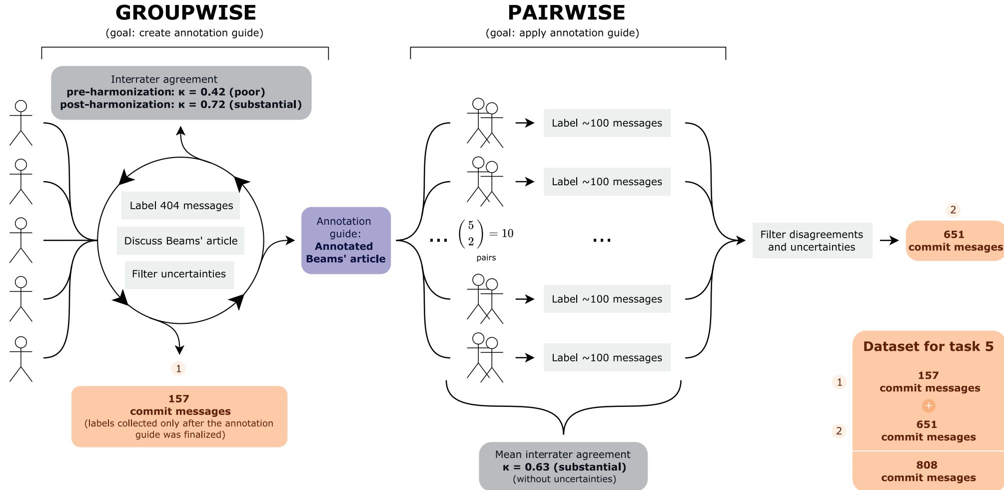

Figure 2. Overview of the labeling process for Task 5.

we continuously monitored κ agreement between experts. We conducted 3 agreement alignment meetings in regular intervals where patterns of disagreements were discussed and resolved. These disagreements were mainly caused by implicit context required for some CMs: Experts found some formulations of CMs to be ambiguous. Especially shorter messages, which naturally might not provide sufficient context, were more often subject to disagreements than longer ones. CMs that were disagreed upon were 171 characters long on average, while CMs that were agreed upon were 48% longer with 253 characters on average. This is probably because the task is heavily context dependent and shorter CMs are more likely to contain implicit context, which can be disagreed upon.

Besides implicit context, another source for disagreements are multiple independent changes in one commit: Some commits are not atomic, i.e., are not following the single responsibility principle (SRP) [2], [29]. This makes it harder to assess whether sufficient reason is given for the change. The experts settle to rate such a CM positively only when the Boy Scout Rule [36] is applied or the message provides reasoning for all of the included logical changes.

V. Evaluation

In this section, we answer our research question **RQ: How well can the CM quality, including semantics and context, be measured with machine learning methods?**. For this, the model performance of various machine learning models used for each task is evaluated via an extensive offline evaluation.

A. Evaluation setting

Table III lists the models we trained and evaluated, consisting of Baseline and of Transformer-based models.

Baselines. For comparison, the following baselines are used: (1) and (2)[5], which are tools specifically for Task 2, and (3) to (7), which have proven to perform well for a variety of NLP tasks, for Task 2-5.

(1) GitCommitBear (GCB) [9],
(2) Bad Commit Message Blocker (BCMB) [12]
(3) Support Vector Machines (SVM) trained with TF-IDF embeddings
(4) Random Forests (RF) trained with TF-IDF embeddings
(5) FastText, a word embedding and classification model that was on par with the performance of deep learning methods as of 2017 [39]
(6) Feed-Forward Neural Network (NN) based on self-trained dense vector word embeddings
(7) Convolutional Neural Network (CNN) with max pooling over time, similar to the architecture proposed by Collobert et al. [48], also based on self-trained dense vector word embeddings.

Transformer-based models. We evaluate models based on BERT [30] that utilize the encoder part of the Transformer architecture [56]. BERT-based models have been pretrained on vast amounts of natural language text. Pretraining facilitates transfer-learning, allowing models to adapt to specific NLP tasks. BERT-based models have been preferred for classification of commit messages in research [32], [33], [51], [55].

We evaluate the original BERT [30] itself, which is pretrained on English Wikipedia and freely available books.

Along BERT, we evaluate 2 models that enhance the architecture, parameters and training data of BERT: RoBERTa [41], a robustly optimized version of BERT, and DeBERTa [35], featuring disentangled attention and an enhanced mask decoder, both trained with more data than BERT - DeBERTa also utilizing stories and data from Reddit, and RoBERTa news articles in addition to that.

We also evaluate DistilBERT [50], a distilled, performant version of BERT that reduces its size by 40% and improves its runtime by 60% by only sacrificing 3% of its language understanding capabilities, rendering it a compelling option

[5]These are the only tools we found that detect the mood of the verb of a CM. They both utilize the POS tagger provided by NLTK [1].

Table III
CLASSIFIERS AND TEXT REPRESENTATIONS OF USED METHODS.

Shorthand	Classifier	Text representation
SVM	SVM	TF-IDF
RF	Random Forests	TF-IDF
FastText	Linear classifier	Word vectors
NN	Feed-forward NN	Word vectors
CNN	Feed-forward NN	Word vectors
BERT	Feed-forward NN	Transformer [56] encoder

for tools that are run locally on the developer's machine.

Additionally, we investigate the performance of models that have been pretrained with data coming closer to commit messages: CodeBERT [32], pretrained unimodally on programming code and bimodally on programming code and programming code descriptions, and SciBERT [27], pretrained on biomedical and computer science papers.

Since our experiments with modified architectures (e.g., using Bi-LSTM) did not yield better performance, we focus on multimodality and fine-tuning various BERT-based models.

Table IV lists for each model the hyperparameters that we optimized, via 10-fold random search cross-validation. When fine-tuning BERT-based models, we vary the epochs in the range of $5, 10, 15$ and the learning rate in the range of 3e-5, 4e-5, 5e-5. After many preliminary experiments with more hyperparameters and broader spectra, we honed in on these ranges, and on the batch-size 8.

Table IV
OPTIMIZED HYPERPARAMETERS FOR EACH MODEL

Method	Hyperparameters
SVM	kernel, degree, C, gamma
RF	number of estimators, max features, max depth, min. samples for internal node split, min. samples required to be at a leaf node
FastText	parameters optimized by [16]
NN	output dimension of embedding layer, number of hidden layers, number of units per hidden layer, L2 regularization parameter λ
CNN	kernel size, output dimension of embedding layer, number of hidden layers, number of units per hidden layer, L2 regularization parameter λ
BERT	learning rate, epochs

B. Evaluation results

Table V summarizes the best 3 BERT-based and the best 3 baseline models for each of our tasks. Besides precision, recall, and F_1, the Table V lists the MCC score (Matthews' correlation coefficient) [42] and its standard deviation. All baseline methods were significantly outperformed by methods based on the BERT architecture. For Task 2, the difference was largest, for Task 4 smallest. The performance of the best three BERT architectures differed by at most 5% for each task. Since CodeBERT was the best (except for Task 4 where

Table V
PERFORMANCE OF THE RESPECTIVE 3 BEST UNIMODAL BERT-BASED AND RESPECTIVE 3 BEST SIMPLE BASELINE MODELS FOR EACH TASK (TASK 1 IS RULE-BASED AND THUS USES NO MODEL THAT NEEDS TO BE EVALUATED), AND BCMB [12] AND GCB [9] FOR TASK 2.

Task	Method	MCC	F_1	Precision	Recall	MCC STD
2	GCB	0.478	0.651	1.000	0.373	0.046
	BCMB	0.563	0.745	0.938	0.551	0.036
	RF	0.622	0.810	0.808	0.815	0.077
	fastText	0.705	0.850	0.890	0.802	0.070
	NN	0.750	0.873	0.889	0.856	0.053
	SciBERT	0.949	0.973	**0.980**	0.967	0.031
	BERT	**0.958**	**0.978**	**0.980**	0.977	0.027
	CodeBERT	**0.958**	**0.978**	0.978	**0.978**	**0.022**
3	fastText	0.709	0.853	0.876	0.809	0.082
	RF	0.718	0.857	0.887	0.811	0.062
	SVM	0.723	0.857	0.910	0.788	0.059
	RoBERTa	0.818	0.906	0.903	0.911	0.053
	DeBERTa	0.823	0.909	0.905	0.914	0.048
	CodeBERT	**0.840**	**0.918**	**0.913**	**0.924**	**0.042**
4	RF	0.822	0.913	0.913	0.912	0.448
	fastText	0.847	0.922	0.955	0.890	0.051
	NN	0.869	0.934	0.934	0.938	0.041
	CodeBERT	0.897	0.946	**0.956**	0.938	**0.022**
	SciBERT	0.898	0.947	0.954	0.941	0.025
	DeBERTa	**0.906**	**0.952**	0.954	**0.949**	0.038
5	CNN	0.529	0.724	0.836	0.720	0.123
	NN	0.532	0.734	0.795	0.783	0.111
	FastText	0.567	0.762	0.823	**0.813**	0.083
	DistilBERT	0.676	0.814	0.823	0.809	**0.069**
	SciBERT	0.686	0.818	0.832	0.809	0.070
	CodeBERT	**0.703**	**0.829**	**0.841**	**0.820**	0.070

DeBERTa was only 1% better), we decided to use CodeBERT as model for all Tasks 2-5, since that simplifies the pipeline and makes it more flexible for future improvements (multimodality, multi-task learning, own pretraining). CodeBERT's higher performance compared to other BERT-based models is likely due to its smaller covariate shift between its pretraining data and our data.

Task 2. We used the additional baseline methods GitCommitBear [9] and Bad Commit Message Blocker [12], which are specialized for this task and therefore listed in Table V in spite of performing poorly. Both are outperformed by all other baseline methods. Their low performance is likely due to the simple n-gram model [8] used to train the POS tagger, which is unable to capture advanced context, and due to the covariate shift between their training and our evaluation data (see [11]).

Task 3 and 4. Task 4 is the only task where CodeBERT is not the best model, but it is only slightly worse (0.006 F_1) than the best. Table V contains our unimodal CodeBERT model that is trained with the CM text alone. In Table VI, we list the performance of our bimodal CodeBERT model for Task 3, which is trained with the CM text and prepended counts of the extensions of changed files (e.g., $md3java1$ for 3 changed Markdown files and 1 changed Java file). Compared to our

Table VI
UNIMODAL (CM) AND BIMODAL (+EXTENSION COUNTS) PERFORMANCE OF CODEBERT FOR TASK 3

Modality	MCC	F_1	Precision	Recall	MCC STD
unimodal	0.840	0.918	0.913	0.924	0.042
+ extension counts	**0.911**	**0.954**	**0.946**	**0.963**	**0.040**

unimodal CodeBERT model for Task 3, bimodality improves the performance of the model significantly, from $F_1 = 0.918$ to $F_1 = 0.954$. Thus, extension counts are a viable additional feature for classification of documentation changes. Our final pipeline incorporates bimodality in Task 3.

We experimented with other categorical and numerical features as part of the textual BERT model input, like file extensions, change lengths and an estimation of the change diff entropy, but only the introduction of extension counts to the model input of Task 3 lead to a significant improvement in the classification performance. Overall, for Task 4, multimodality did not improve performance.

Task 5. Compared to other tasks, CodeBERT won most clearly for Task 5 with an F_1 difference of 0.011 compared to the second best, SciBERT. As for Task 4, multimodality did not improve performance for Task 5.

Being the most challenging task, we analyze the performance in more detail: Firstly Table VII summarizes F_1, precision, and recall for both positive (minority class, Task 5 ratings < 2.5) and negative (majority class, Task 5 ratings ≥ 2.5) class for our best model, CodeBERT. Secondly, Table VIII compares the performance of the best model (BERT) from Tian et al. [55] and our best model (CodeBERT). We evaluate both models on our own dataset, due to the deficits of Tian et al.'s dataset, see Sec. II. Our model is evaluated using 10-fold cross-validation, while their model was trained on their own dataset and then evaluated on our dataset. Comparing both models on our dataset, our model has a 35% better F_1 score, 32% better precision, and 53% better recall. Evaluating their model on our dataset instead of their dataset, their (see [55]) positive class F_1 score drops from 77.6% to 73.6% (5%), their negative class F_1 score from 73.9% to 61.5% (17%). This shows that their model does not perform well for other programming languages than Java, domains outside high quality open-source web communication, a semantically meaningful interpretation of rule R7 (see Sec. II), and is thus unsuitable for general use. In contrast, our approach does not have these limitations, as the F_1 score of 82.9% shows. The performance can likely be improved further by increasing Task 5's dataset (see Sec. VI-C), as the plot of the predictive performance, depending on the dataset size, shows in Fig. 3.

Task 5: error analysis. Since Task 5 is the most challenging and yields the lowest performance among our tasks, we conduct an error analysis: Firstly, we see in Table VII that our model is weakest on the positive class recall with 0.82. With a FNR of 0.18, our model classifies rather lax and predicts 18% of bad commit messages as good. But for a quality assurance tool, being too lax is better than being too strict since a too

Table VII
PERFORMANCE OF CODEBERT FOR TASK 5.

Class	F_1	Precision	Recall
Negative	0.872	0.866	0.880
Positive	0.829	0.841	0.820

Table VIII
PERFORMANCE COMPARISON WITH TIAN ET AL.'S PRETRAINED CLASSIFIER [55] ON OUR DATASET

Method	F_1	Precision	Recall	Accuracy
Tian et al. (BERT)	0.615	0.658	0.577	0.759
We (CodeBERT)	**0.829**	**0.841**	**0.820**	**0.854**

high FPR leads to noise, manual work, distrust, and eventually rejection of the tool. Nonetheless we look into the cause for our lax model: For Fig. 4, we conduct another 10-fold cross-validation to create confusion matrices on the *random pool* resp. the *good pool* (see Sec. IV-A). As expected, the FNRs are higher than the FPRs. The model has a significantly higher TPR in the *random pool* (80%) than in the *good pool* (67%), likely due to many more obviously bad CMs in the *random pool*. As consequence from the high FNR in the *good pool*, the overall positive class recall is relatively low. So the overall positive class recall would be better with our data pipeline sampling the training set only from the *random pool* instead of evenly from the *random pool* and *good pool*. We believe even sampling was the right choice because our model should also be helpful for teams that already take pride in writing high quality commit messages. With a training set from the *random pool* only, the model would have a harder time to generalize to the *good pool*, and Fig. 4 shows that the model with even sampling already performs worse on the *good pool* than on the *random pool*. All in all, we conclude from Fig. 4 that the model performance for the *good pool*, though reasonable, could be improved by labeling and sampling more CMs from the *good pool*.

Runtime performance. The CodeBERT models are about 500 MB in size and evaluating a sentence takes about 7

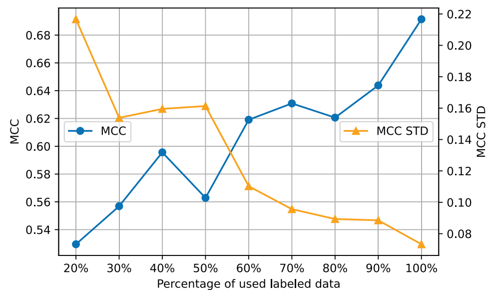

Figure 3. Effect of data size on the predictive performance of our best model for Task 5 where 100% corresponds to the 808 commits in our sample evaluated by a 10-fold CV.

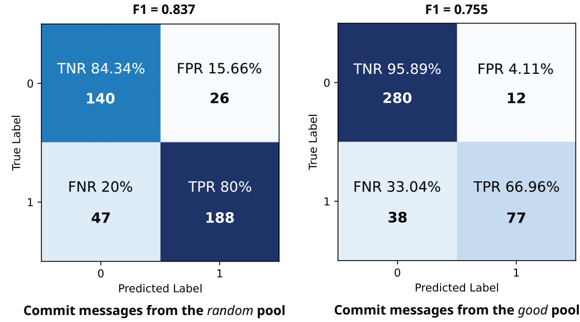

Figure 4. Performance of our CodeBERT trained and evaluated with 10-fold CV, with evaluation results separated for commits from the *good* and *random* pools.

seconds on a modern i7-1068NG7 CPU without utilizing GPU support. This may not be sufficient for a satisfactory user experience for real-time evaluation while the CM is being typed. When exporting the model to the ONNX [14] format and thus profiting from hardware optimizations, the runtime of a single model prediction in the pipeline on the CPU can be reduced significantly to 0.02 seconds. Thus, we use ONNX.

In conclusion, given the high predictive performance of $F_1 = 82.9\%$ for our CodeBERT model for Task 5, our answer to **RQ: How well can the CM quality, including semantics and context, be measured with ML methods?** is: sufficiently high for practical use.

For illustrative purposes, we demonstrate the application of our pipeline to a random sample of 5,000 CMs in Figure 5.

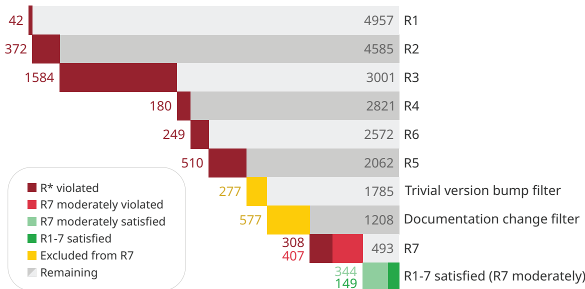

Figure 5. Our pipeline classifying 5,000 randomly sampled commits

Contribution 3: Using our datasets, we fine-tune current NLP methods. They achieve state-of-the-art and can measure CM quality – including semantics and context – sufficiently well for practical use, answering our research question positively.

VI. Discussion

A. Predictors of CM quality

We use our tool to assess various repositories (repos for short) and look into multiple aspects of software engineering to investigate predictors of CM quality, with linear regression and Pearson correlations (since outliers occurred rarely and in those cases the Spearman correlation coefficient was only insignificantly different). All reported correlation coefficients have $p < 0.05$.

Commit-based predictors. Firstly, we scrape a large *Commit Guru dataset*[2] from [23]. It comprises 482 repos, each with its full commit history, totaling 4,103,489 commits, and each with many per-commit metrics. Commit Guru [49] derives these metrics as estimates for aspects like the number of developers involved, their experience, whether the commit introduces a bug (*bug-inducing*), and various aspects regarding time. Since there are 16 numerical metrics, we firstly analyze their relationship with R7 ratings with an OLS multiple linear regression: it is significant and the per-commit metrics do correlate with R7 ratings (*F-statistic* $= 17530$ with $p = 0$, each predictor's $p < 0.05$), but with very high variance and each individual per-commit metric only has a very weak correlation with R7 ratings ($R^2 = 0.068$, prediction intervals cover the full R7 rating scale). We perform another OLS multiple linear regression on per-repo averages across all 482 repos, to check whether these per-repo metrics, e.g. the average number of developers involved in a repo, have a stronger effect on the averaged R7 ratings. This regression is also significant, and at least 6 per-repo metrics correlate with R7 ratings (*F-statistic* $= 10.79$ with $p = 0$, and $p < 0.05$ for 6 predictors), but still with relatively high variance and each individual per-repo metric has no or a very weak correlation with R7 ratings ($R^2 = 0.258$, prediction intervals cover half of the R7 rating scale). These weak correlations might be due to software engineering being complex, with many interacting explanatory variables – regressions have very high condition numbers of 1.01e+16 (per-commit) resp. 2.25e+05 (per-repo). Consequently, we will only look at a few specific relationships in the *Commit Guru dataset*. A more elaborate analysis on these predictors is future work.

Correlations with commit-based predictors. Firstly, we investigate the per-commit relationship between R7 ratings and *bug-inducingness*, which is estimated with an SZZ-like algorithm [54] that detects bug-fixing commits via keyword detection in CMs, from which bug-inducing commits are derived via git blame [49]. As expected after our regression analysis, we only get a weak correlation, surprisingly a positive one. Computing the per-repo relationship instead, i.e. the correlation between the percent of *bug-inducing* commits and average R7 ratings leads to a similar result, as does the relationship between commit's *bug-inducingness* and their average R7 ratings so far. These are likely due to confounding variables: writing better commit messages leads to better keyword detection and thus better detection of buggy commits; and more complex environments cause more bugs, but also demand better CMs (see moderate correlation $r_{\text{systemsPL}}$ below).

Thus we consider other, per-repo predictors. Firstly, we compute the correlation between the per-repo average R7 rating and the lifetime of the repo, measured by the time

Table IX
PLS WITH AVERAGE R7 SCORE, AGE, AVERAGE AGE OF DEVELOPERS

Main PL	Score	Age of PL	Age of Dev
C	3.34	51	31.7
C++	3.14	38	32.0
Perl	2.89	35	40.3
Ruby	2.80	28	34.2
PHP	2.77	28	32.5
Objective-C	2.76	38	34.6
Python	2.75	32	30.7
JavaScript (JS)	2.68	27	32.5
Java	2.60	28	32.3
Go	2.58	14	32.1
Scala	2.51	18	33.3
Lua	2.43	29	33.7
TypeScript (TS)	2.37	10	30.6
Swift	2.34	9	30.8
C#	2.15	23	33.8
CoffeeScript	2.12	13	-

span between the repo's first and last commit[6]: we get a moderate positive correlation $r_{\text{lifetime}} = 0.30$. If r_{lifetime} is due to a causal relationship, it is not from lifetime to R7 ratings, since the R7 ratings are roughly constant over time, but from R7 ratings causing more maintainable repos and thus higher sustainability, i.e. higher lifetime. Secondly, we consider team size as predictor and compute a weak positive correlation $r_{\text{team}} = 0.28$ between the average R7 ratings and the average number of developers, likely due to the increased necessity of communication in larger teams.

Correlation with more general predictors. Having at most moderate correlations on commit-based metrics, we consider more general predictors: programming language, developers, companies. We use our *source dataset* (see Sec. IV-A), which comprises 1,700 repos with 100 commits each, covering 17 programming languages (PLs) and many companies.

A repository has the main PL P if more than 50% of the files changed by its commits are in P. Table IX lists the main PL, the average R7 rating, the age of the PL, and the average age of the developers of the PL [21]. This grouping leads to the following correlations:

- projects using younger PLs have lower average R7 rating, with a strong correlation $r_{\text{agePL}} = 0.83$
- projects in a systems PL (C, C++, Go) have a higher average R7 rating, with a moderate correlation $r_{\text{systemsPL}} = 0.58$, likely because developers know that low-level programs interface with a complex environment outside of the software system and thus demand thorough descriptions of that complex context in the CMs.

To investigate whether the strong correlation r_{agePL} is caused by developer demographics, we estimate the experience of a developer d at the time they created commit c by getting (a) the time span between d's first GitHub commit and c, and (b) the number of GitHub commits of d up to c. On our dataset, there is only a weak positive correlation $r_{\text{experienceDev}} < 0.08$ between R7 ratings and (a) resp. (b)[7]. Additionally, the average developer age estimated for each PL P (see Table IX) has a weak correlation $r_{\text{ageDev}} = 0.13$ with P's average R7 rating.

[6]The average over all repos is 8.2 years, its standard deviation 5.2 years.

Table X
(GROUPS OF) CORPORATIONS WITH THEIR CM QUALITY

Corp	Uber	Apple	Twtr	Meta	Goog	Nflx	MS
Score	3.92	3.56	3.43	3.38	3.20	2.70	2.68

Group	Big-4	Big-5	Big-tech	Non-big-tech
Score	3.37	3.19	3.24	2.67

Is the strong correlation r_{agePL} caused by team culture? Lacking a metric for company culture, we group the commits into their company workplace, i.e. owner, of the corresponding repository: The average R7 rating for each owner is listed in Table X (Big-4 is Amazon, Apple, Meta, Alphabet (Goog); Big-5 is Big-4 and Microsoft (MS); Big-tech is Big-5 and Netflix (Nflx), Snap, Twitter (Twtr), and Uber; everything else is Non-big-tech). This shows a strong relationship between company culture and CM quality.

In summary, a CM quality assessment with our tool is not only useful for improving CMs, but can also reveal other software engineering aspects, e.g. about the developer culture. The per-repo correlations with average R7 ratings are

- for PLs: a strong correlation of $r_{\text{agePL}} = 0.83$ with the main PL of the repo, and a moderate correlation $r_{\text{systemsPL}} = 0.58$ with the system level of the PL,
- for developers: weak correlations of $r_{\text{ageDev}} = 0.13$ with the estimated average age of the developers and $r_{\text{experienceDev}} \leq 0.09$ with the estimated experience of the developers, and a moderate correlation of $r_{\text{team}} = 0.28$ with the team size,
- for sustainability: a moderate correlation of $r_{\text{lifetime}} = 0.30$ with the lifetime of the repo.

B. Threats to validity

Threats to construct validity. Fig. 3 indicates that training with more data can improve the predictive performance of our model for Task 5, which we already started working on (see next section). Our data collection was limited by the experts' availability and the high labeling effort for Task 5.

Threats to internal validity. CodeBERT's text input is limited to 512 word(-parts) [53], which can cause attrition or deformation (e.g., clipping). But this happened only for extremely long CMs, in less than 0.4% of the cases.

Threats to external validity. Some developers neglect writing high quality CMs, arguing that their workflow is issue tracker centric. But software maintainability is improved by having both issues and CMs of high quality. The CM

[7]This is in line with the weak correlation $r = 0.09$ between R7 ratings and the per-repo developer experience metric of the *Commit Guru dataset*.

history is independent of the issue tracking tool, closer to the code and developer, and accessible by many tools, e.g. IDEs. Furthermore, there is always a one-to-one relationship between commits and CMs, but this is often not the case between commits and issues. Finally, many CMs in our dataset had dead PR- or issue-links, pointed to issues without any useful text or to issues with information hidden in too much text or behind links (see e.g. Table I). Thus the git history should contain the relevant information directly.

C. Future Work

Additional labels. We started to annotate further CMs with additional labels, covering the following aspects: "why" sufficient, "what" sufficient, "how" not distracting, CM matches code change. Differentiating these aspects is quite complex and time consuming, but we plan to gain further insights and performance improvements from having more data and more fine-grained labels.

Parallel rule checking. Our pipeline processes a CM sequentially through our classification models. A parallel architecture would enable faster evaluation of high quality CMs, and multiple warnings at once for low quality ones. However, it would have required experts to also label CMs for the challenging rule R7 that fail the simpler rules R1 to R6 for various reasons. But many of those CMs are hard for the experts to understand and label because they have confusing formatting (filtered out by Task 1), follow project-specific conventions (filtered out by Task 3 and 4), or text that does not clarify whether it is about the change or about the context and the way things worked before the change (filtered out by Task 2). Due to the higher variance of those CMs, the model for R7 would also need more training data to achieve a certain performance.

User study. We plan a user study to identify possibilities for improvement. There is already interest from several medium and large companies to participate.

Pretraining BERT. CodeBERT likely performs better than other BERT-based models because it has a smaller covariate shift between its pretraining data and our fine-tuning data. There is some covariate shift left even for CodeBERT, so pretraining a BERT-based method with pairs of CMs and matching code diffs will likely further improve the performance, by achieving advanced comprehension of code diffs and thus improve the predictive performance for our tasks. But pretraining is expensive, as well as data- and time-consuming, and thorough experiments on pretraining tasks on commits are needed.

VII. Conclusion

The quality of CMs is crucial for software maintenance and evolution. Thus, we considered how well CM quality can be assessed, and created a framework for automatically assessing CM quality based on several criteria, using state-of-the-art ML methods. Our contributions are all open-sourced[2].

Specifically, we developed and provided large **datasets** of labeled CMs for the detection of imperative mood (Task 2), of version bump (Task 3), and of documentation changes (Task 4). Furthermore, we designed a high-effort dataset of labeled CMs for evaluating their semantic quality based on context and meaning (Task 5 for Chris Beams Rule 7 about "what and why vs. how"). These datasets are also useful for benchmarking and other tasks, such as research about CM generation. We then provided thorough **evaluations** of our full-fledged framework, consisting of 4 classification tasks. To this end, we considered 7 baselines and 5 deep learning-based models. The best performing models (BERT-based trained on our datasets) set the state-of-the-art for all our classification tasks, with the following F_1 scores: 97.8% for Task 2 (formerly 74.5%), 95.4% for Task 3, 95.2% for Task 4, 82.9% for Task 5 (formerly 61.5%). Finally, we created our framework as a **tool** for practitioners, which can be run locally or in the cloud, and comprises a pipeline to automatically assess the CM quality on all levels (format, syntax, semantics), checking all rules of the CM guideline by Chris Beams [26], the most popular one.

The evaluation demonstrates that our open-source framework can automatically assess the quality of CMs, including semantics and context, sufficiently well for practical use. Our framework can be used for assessing projects and company culture, or integrated in a software development process. For instance, as an IDE plugin for immediate validation while a CM is being written, or as commit hook [22] for a simple solution that still prevents commits with low quality messages from being merged, or in the CI as an automatic CM quality reviewer. There is already interest from medium and large companies to participate in our planned user study. A CI integration in the background, e.g., of GitHub projects with GitHub Actions [17], can lead to a broad adoption and thus to a shift towards more maintainable and faster evolving software.

Acknowledgment

We thank Jochen Krause, CEO of Innoopract Informationssysteme GmbH, for his big support with knowledge and investment of 5 experts for labeling.

Furthermore, we acknowledge support by the state of Baden-Württemberg through bwHPC and by KI-Lab Lübeck for provided infrastructure.

References

[1] Natural Language Toolkit — NLTK. https://www.nltk.org/, 2001.

[2] Who-T - On commit messages. https://who-t.blogspot.com/2009/12/on-commit-messages.html, 2009.

[3] angular/angular - CONTRIBUTING.md. https://github.com/angular/angular/blob/master/CONTRIBUTING.md#-commit-message-format, 2011. Documentation published in 2015 but the convention has been in use since 2011 in the Angular.js project [4].

[4] First usages of the Angular.js commit convention [3]. https://github.com/angular/angular.js/commits/8bae2a5e, 2011.

[5] erlang/otp - Writing good commit messages. https://github.com/erlang/otp/wiki/writing-good-commit-messages, 2012.

[6] 5 Useful Tips For A Better Commit Message. https://thoughtbot.com/blog/5-useful-tips-for-a-better-commit-message, 2013.

[7] Standard to follow when writing git commit messages. https://stackoverflow.com/questions/15324900/standard-to-follow-when-writing-git-commit-messages, 2013.

[8] nltk.tag.perceptron - NLTK documentation. https://www.nltk.org/_modules/nltk/tag/perceptron.html#PerceptronTagger, 2015.

[9] coala/coala-bears: CommitBear.py, check_imperative. https://github.com/coala/coala-bears/blob/7d21a59/bears/vcs/CommitBear.py#L225, 2016.

[10] Gitmoji - An emoji guide for your commit messages. https://gitmoji.dev/, 2016.

[11] NLTK commit message POS tagging issues. https://github.com/coala/coala-bears/issues/243#issuecomment-204484597, 2016.

[12] platisd/bad-commit-message-blocker - Inhibits commits with bad messages from getting merged. https://github.com/platisd/bad-commit-message-blocker, 2016.

[13] Conventional Commits. https://www.conventionalcommits.org/en/v1.0.0/, 2017.

[14] ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx, 2017.

[15] Working with the kernel development community – Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/v4.14/process/submitting-patches.html, 2017.

[16] fastText - Automatic hyperparameter optimization. https://fasttext.cc/docs/en/autotune.html, 2019.

[17] GitHub Actions. https://github.com/features/actions, 2019.

[18] TimescaleDB's Git Hook To Check Chris Beams' Rules. https://github.com/timescale/timescaledb/blob/master/scripts/githooks/commit_msg.py, 2019.

[19] Commit Styleguide of NIST's project Dioptra (Test Software for the Characterization of AI Technologies). https://pages.nist.gov/dioptra/dev-guide/contributing-commit-styleguide.html, 2020.

[20] Fix Sass + CSS Modules #3186. https://github.com/withastro/snowpack/pull/3186, 2021.

[21] Programming Languages by Age. https://www.reddit.com/r/dataisbeautiful/comments/rkv4gy/comment/hpbxy1l/?utm_source=share&utm_medium=web2x&context=3, 2021.

[22] The ITK Software Guide. https://itk.org/ItkSoftwareGuide.pdf, 2021.

[23] Ask the Commit Guru. http://commit.guru, 2023.

[24] Kapil Agrawal, Sadika Amreen, and Audris Mockus. Commit quality in five high performance computing projects. In *2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*, pages 24–29. IEEE, 2015.

[25] Harry Alberto and Carpio Salvatierra. *A Proposal of Automatic Quality Evaluator for Git Commit Messages*. PhD thesis, Universidad Politécnica de Madrid, 2019.

[26] Chris Beams. How to Write a Git Commit Message. https://chris.beams.io/posts/git-commit/, 2014.

[27] Iz Beltagy, Kyle Lo, and Arman Cohan. SciBERT: Pretrained Language Model for Scientific Text. In *EMNLP*, 2019.

[28] Scott Chacon and Ben Straub. Pro Git: Distributed Git - Contributing to a Project. https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project, 2014.

[29] Kuljit Kaur Chahal and Munish Saini. *Developer dynamics and syntactic quality of commit messages in OSS projects*, volume 525. 2018.

[30] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1(Mlm):4171–4186, 2019.

[31] Khaled El Emam. Benchmarking Kappa for Software Process Assessment Reliability Studies. *Empirical Software Engineering: An International Journal*, 4:113–133, 1998.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. pages 1536–1547, 2020.

[33] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, jul 2021.

[34] Ken Gu and Akshay Budhkar. A Package for Learning on Tabular and Text Data with Transformers. pages 69–73, 2021.

[35] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. DeBERTa: Decoding-enhanced BERT with Disentangled Attention. pages 1–21, 2020.

[36] Kevlin Henney. *97 things every programmer should know: collective wisdom from the experts*. O'Reilly Media, 2010.

[37] Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiang-Ping Chen, Zi-Bin Zheng, and Ming-Dong Tang. Learning human-written commit messages to document code changes. *Journal of Computer Science and Technology*, 35(6):1258–1277, 2020.

[38] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically Generating Commit Messages from Diffs using Neural Machine Translation. *arXiv*, pages 135–146, 2017.

[39] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017 - Proceedings of Conference*, 2:427–431, 2017.

[40] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[41] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. (1), 2019.

[42] B. W. Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *BBA - Protein Structure*, 405(2):442–451, oct 1975.

[43] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. *Conference on Software Maintenance*, (January 2000):120–130, 2000.

[44] Tiago Oliveira Motta, Rodrigo Rocha Gomes Souza, and Claudio Sant'Anna. Characterizing architectural information in commit messages: An exploratory study. *ACM International Conference Proceeding Series*, (ii):12–21, 2018.

[45] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. Coregen: Contextualized code representation learning for commit message generation. *Neurocomputing*, 459:97–107, 2021.

[46] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[47] Tim Pope. A Note About Git Commit Messages. https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html, 2008.

[48] Ronan Collobert. Natural Language Processing (Almost) from Scratch. 2017-Janua:2493–2537, 2017.

[49] Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 966–969, 2015.

[50] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv*, pages 2–6, 2019.

[51] Muhammad Usman Sarwar, Sarim Zafar, Mohamed Wiem Mkaouer, Gursimran Singh Walia, and Muhammad Zubair Malik. Multi-label Classification of Commit Messages using Transfer Learning. *Proceedings - 2020 IEEE 31st International Symposium on Software Reliability Engineering Workshops, ISSREW 2020*, pages 37–42, 2020.

[52] Thomas Schweizer. Université de Montréal Towards Using Fluctuations in Internal Quality Metrics to Find Design Intents par Thomas Schweizer. 2020.

[53] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *54th Annual Meeting of the Association for Computational Linguistics, ACL 2016 - Long Papers*, 3:1715–1725, 2016.

[54] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[55] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? 2022.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017-Decem(Nips):5999–6009, 2017.

[57] Sarim Zafar, Muhammad Zubair Malik, and Gursimran Singh Walia. Towards Standardizing and Improving Classification of Bug-Fix Commits. *International Symposium on Empirical Software Engineering and Measurement*, 2019-Septe, 2019.