e-Informatica Software Engineering Journal, Volume 19, Issue 1, 2025, pages: 250102, DOI: 10.37190/e-Inf250102

ACoRA – A Platform for Automating Code Review Tasks

Mirosław Ochodek*, Miroslaw Staron**
*Institute of Computing Science, Poznań University of Technology
**IT Faculty, Chalmers University of Technology | University of Gothenburg
miroslaw.ochodek@put.poznan.pl, miroslaw.staron@gu.se

Abstract

Background: Modern Code Reviews (MCR) are frequently adopted when assuring code and design quality in continuous integration and deployment projects. Although tiresome, they serve a secondary purpose of learning about the software product.
Aim: Our objective is to design and evaluate a support tool to help software developers focus on the most important code fragments to review and provide them with suggestions on what should be reviewed in this code.
Method: We used design science research to develop and evaluate a tool for automating code reviews by providing recommendations for code reviewers. The tool is based on Transformer-based machine learning models for natural language processing, applied to both programming language code (patch content) and the review comments. We evaluate both the ability of the language model to match similar lines and the ability to correctly indicate the nature of the potential problems encoded in a set of categories. We evaluated the tool on two open-source projects and one industry project.
Results: The proposed tool was able to correctly annotate (only true positives) 35%–41% and partially correctly annotate 76%–84% of code fragments to be reviewed with labels corresponding to different aspects of code the reviewer should focus on.
Conclusion: By comparing our study to similar solutions, we conclude that indicating lines to be reviewed and suggesting the nature of the potential problems in the code allows us to achieve higher accuracy than suggesting entire changes in the code considered in other studies. Also, we have found that the differences depend more on the consistency of commenting rather than on the ability of the model to find similar lines.

Keywords: Code Reviews, Continous Integration, BERT, Machine Learning

1. Introduction

Modern Code Reviews (MCR) [1, 2] is a common practice in continuous integration and deployment companies. A modern code review is a practice that evolved from software inspections advocated by Fagan et al. [3] already in 1976, but it adapts to modern tools and the ability to peer review smaller segments of code (commits) pushed by developers to the main branch of the code. MCR is integrated into modern software development pipelines and all leading configuration management platforms enable this way of working. Git and Gerrit [4, 5] are two examples of such tools, where the developers can review other's code before it is integrated with the main branch.

© 2025 The Authors. Published by Wrocław University of Science and Technology Publishing House.
This is an open access article under the CC BY license international.
Submitted: 18 Jan. 2024; Revised: 5 Aug. 2024; Accepted: 2 Sep. 2024; Available online: 3 Oct. 2024

PREVIEW

Although MCR is a lightweight process compared to the original inspection process of Fagan, it is still a tiresome process and can result in delays when delivering the product [6]. It is also a process known to miss important quality issues [7]. To address the problem, the majority of current research efforts are targeted towards either eliminating this activity by automated code repairs [8], improving the tools used for code reviews [9], or even predicting which lines of code should be reviewed manually [10, 11].

However, one of the main limitations of the automated code repair activities is the low success rate (ca. 30% at best, [8]). The major drawback of the automated suggestion for which code fragments to review is the lack of information on why and what should be reviewed exactly in that code fragment. Furthermore, MCR has secondary goals in addition to quality assurance. It is often perceived as a good way of onboarding developers into new projects and learning within the team [12, 13]. Therefore, in this paper, we address the research problem of:

To which degree can we suggest relevant review guidance for a given code fragment based on historical data?

We address this question by designing and constructing an automated code review assistance platform—`ACoRA`. The platform is based on the idea that a programming language can be treated as a natural language from the perspective of machine learning language models [14]. It employs a Transformer-based language model [15] to search for lines of code similar to those under review that were previously commented on. Later, it analyzes the comments to highlight the aspects of code on which the reviewer should focus while reviewing a given code fragment. It performs a multi-class / multi-label classification according to the proposed taxonomy [16] and aggregates the results over the comments for similar lines to guide the reviewers' focus. `ACoRA` can be integrated with MCR tools to learn from the previous reviews to be able to suggest what should be reviewed in a given code fragment.

We use design science research as our methodology as prescribed by [17] and evaluate `ACoRA` on both open-source projects and together with an industrial partner. The results show that we can suggest completely correct recommendations (only true positives) in 35%–41% of the fragments and partially correct in 76%–84% of the fragments. The results are better than the results of similar studies (e.g., suggesting code that repairs a defect or suggesting a review text itself).

The paper is structured as follows. Section 2 summarizes the most relevant related research. Section 3 describes the `ACoRA` platform and Section 4 details our research methodology. Sections 5 and 6 present and discuss the results of evaluating `ACoRA` and Section 7 discusses the threats to validity. Finally, Section 8 presents the conclusions and outlines the further work in this area.

2. Related Work

The field of using natural language processing models for programming tasks is developing rapidly. In this section, we provide the current and the most relevant related research in this area.

PREVIEW

2.1. Natural Language Processing models applied to code

"On the naturalness of software" by [14] is a seminar work that started a number of research directions in using machine learning for programming tasks. This paper shows that there exist several approaches for using natural language processing machine learning models in software engineering, with a focus on such tasks as program repair or defect finding.

In fact, this field became very popular and a survey by [18] formalized a hypothesis about the naturalness of programming languages: *"The naturalness hypothesis. Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools."* This hypothesis provided a foundation to classify approaches to processing programming languages, but the most important contribution of this work is the classification of the tasks where the hypothesis is used (at least initially, when it was formulated): code-generating models, representational models of code, and pattern mining models. The paper also reviewed each of these application areas and found a significant number of models and applications. For example, for the code-generating models, they found 49 studies, and the number is certainly higher today. Our work, however, is focused on studying the last area—pattern mining of code, although focused on applying these techniques to a specific task—code review support. That area showed a significantly smaller set of studies—10 studies in 2018.

2.2. Using language models for code repair and generation tasks

One of the first models, which is used in several studies, is the representation of code using information about the program's abstract syntax tree—code2vec, as introduced by [19]. The underlying concept behind code2vec is that a program's code should be represented as a feature vector using information taken from the grammar of the programming language. The feature vector carries information about whether a code fragment is a definition of a function, a single statement, or even a specific type of token. The approach has been demonstrated to work well when translating programming languages or finding meaningful synonyms. However, it has one disadvantage. Namely, it requires the program to compile, which is quite problematic in industrial contexts as well as in the context of MCRs. Since the focus of MCR is, per definition, one commit, then the set-up of the entire code2vec alongside compilers can be problematic, as we found in our previous studies [6].

As opposed to the grammar-based language models, a new trend emerged when BERT models [20] showed significant progress in the area of natural language processing, in particular in translation between programming languages. One of the first BERT-based models in this area is CodeBERT designed by [21]. The model is pre-trained for programming languages like JavaScript or Python along with the English natural language. The model supports translations between a natural language and a programming language for such tasks as writing programs based on natural language commands or summarization of programs in natural languages (documentation generation).

Program generation and translation of natural language to programs are also the base tasks for the TransCoder models presented by Facebook Research. An example of such a task is the deobfuscation task, where the models change identifiers in the program to ones that are more meaningful for software developers, as presented by [22]. The performance of the model is impressive and in the best-case scenarios achieves an accuracy of 67.6%.

PREVIEW

AlphaCode as presented by [23], is another prominent example of current state-of-the-art models in program generation. The model is trained to solve problems from programming competitions and uses the same technology. The application of the model demonstrates that it can generate programs based on natural language specifications and also ranks in the top 54.3% compared to ca. 5,000 participants. The main difference of this model is that it is trained to solve "artificial" programming tasks, which are not the same as software engineering tasks in the industry. Our work is based on the same principles as AlphaCode (using transformer models) but aligned with the industrial needs and requirements.

However, the accuracy depends on the dataset and the task of the model. An example of a problem, which is significantly more relevant to the industrial context is program repair. There, the newest models, such as Review4Repair [24], can achieve an accuracy of above 30%, which is still a significant improvement from the previous models. Review4Repair solves the task of fixing a given defect based on finding and adapting code fragments from Git. The problem is significantly more difficult than deobfuscation or program generation since the new code fragment has to fit in the existing context. A similar approach was followed by Tufano et al. [25] who studied the possibility of adapting Text-To-Text Transfer Transformer (T5) to either automatically suggest changes in the code under review or to generate such changes based on the reviewer comments. These ways of using language models are the most similar to our approach, with the difference that we do not generate code fragments but guide the focus of the reviewers by indicating lines of code similar to the previously commented ones with hints on potential reasons for comments. Thus, we solve a modified version of this problem. As opposed to Review4Repair, we provide a recommendation for a software developer, who needs to react, rather than providing a solution that needs to be automatically approved (e.g., through testing).

Finally, the latest commercial achievement in this line of research is the GitHub Copilot[1], which is based on OpenAI's GPT-4 model. GitHub Copilot tool can provide both suggestions for new code fragments based on natural language comments of what the code should do, and based on the previous code that has been written (code completion). Our model solves a simpler problem but is trained on a codebase selected by the software developers, which does not pose any legal issues, as they can choose to use the model only on their previous code.

2.3. Modern Code Reviews

The focus of our work, i.e., code review processes in the continuous integration/deployment context, has industrial roots. The MCR process is effort-intensive and can vary in quality. One of the seminal papers about MCR, and its industrial role, is the study of code reviews at Google [26]. Among other findings, [26] presents identifying code ownership and readability as important factors in the process of code review. They have also found that code reviews should be done on smaller parts of the code to make the process faster, which has implications for the technology used to support the code reviews. The smaller fragments of code require the models to use non-grammar-based approaches. It also poses requirements for being specific when providing recommendations. These implications have also been identified in our previous studies [6] and therefore `ACoRA` generalizes the review suggestions as well as provides recommendations on arbitrary code fragments, e.g., individual lines.

[1]https://copilot.github.com

PREVIEW

One of the fundamental challenges when applying machine learning to code reviews is our ability to understand what a good code review is. [27] studied the concept of code review and source code change from the perspective of how a good change, or review, is defined in the literature. They verified their studies based on industrial practice. Small size of the change, clear context, and relevant suggestions are some of the identified factors. In `ACoRA`, we follow these findings and provide generalized suggestions about the potential nature of the problem and its context (examples of similar lines of code previously commented on and the comments) to help the developers make the most of the suggestions.

One of the challenges we encounter in our work is the ability to characterize and categorize code review comments. [28] studied a set of code review comments from over 2,000 software developers. Their study used pre-defined, fine-grained categories of code reviews and achieved an accuracy of 63.9%. This shows that the accuracy of the `ACoRA`'s `BERT4Comments` model (above 86%) is in line with other models performing automatic classification of pull-request comments, e.g., [29, 30, 31].

Continuing in this area, the state-of-the-art models for review recommendation use ensembles and several matches to provide a good recommendation. For example, automating code review tool CORE, presented by [32], uses a corpus of 57,000 code review comments and obtains results at the level of 11% (recall for one suggestion) and 48% (recall for ten suggestions). The results are improvements of two orders of magnitude compared to the previous tools. The low recall for the first suggestion, however, can be linked to the fact that the CORE tool generates natural language suggestions, i.e., a text for the comment. In order to reduce the complexity, `ACoRA` suggests categories of problems rather than generating the text. Our results outperform CORE's recall for a single suggestion.

Instead of providing suggestions for code reviews in general, there are tools that focus on specific types of suggestions. An example of such a tool is the RAID tool, as presented by [33]. The tool focuses on identifying code refactoring opportunities based on analyzing code reviews in MCR. The tool results in significant improvements in the code base, e.g., by reducing the size of the codebase. It also, like `ACoRA`, uses software developers in the loop to make the assessment of the quality of the recommendations.

Tufano et al. [34] studied the strengths and weaknesses of contemporary code-review automation approaches. They identified three types of code-review automation tasks, i.e., code-to-comment, code & comment-to-code, and code-to-code. The first one, code-to-comment, is about generating review comment text for a piece of code under review that would match the comment made by a human expert reviewer. `ACoRA` partially fits into this category. However, instead of generating a comment, we aim to guide the focus of the human reviewers, first to the lines that might need their attention, secondly by suggesting what they should focus on when reviewing the code, and finally by showing examples of comments for similar lines. By doing so, we limit the weaknesses of similar tools that use historical code-review comments to generate review comments, such as CommentFinder [35], since we do not narrow the recommendation to a proposal of a single comment but rather guide the focus of the reviewer.

Finally, the field of MCR has been developing rapidly, and there are several systematic reviews on the topic, e.g., by Badampudi et al. [36, 37], by Davila and Nunes [38], and by Cetin et al. [39]. We can summarize the current state-of-the-art as being focused on either understanding the process of code reviews or providing tool support. Our work contributes to the latter, in particular by creating a support tool. We automate the process and support learning (onboarding new project members, solution discussions) rather than replacing core reviewers.

PREVIEW

3. Automated Code Review Assistant platform

Automated Code Review Assistant[2] (ACoRA) is a platform for automated review recommendations based on historical code reviews. In general, it covers two processes: a *configuration process* and a *recommendation process* (see Figure 1).

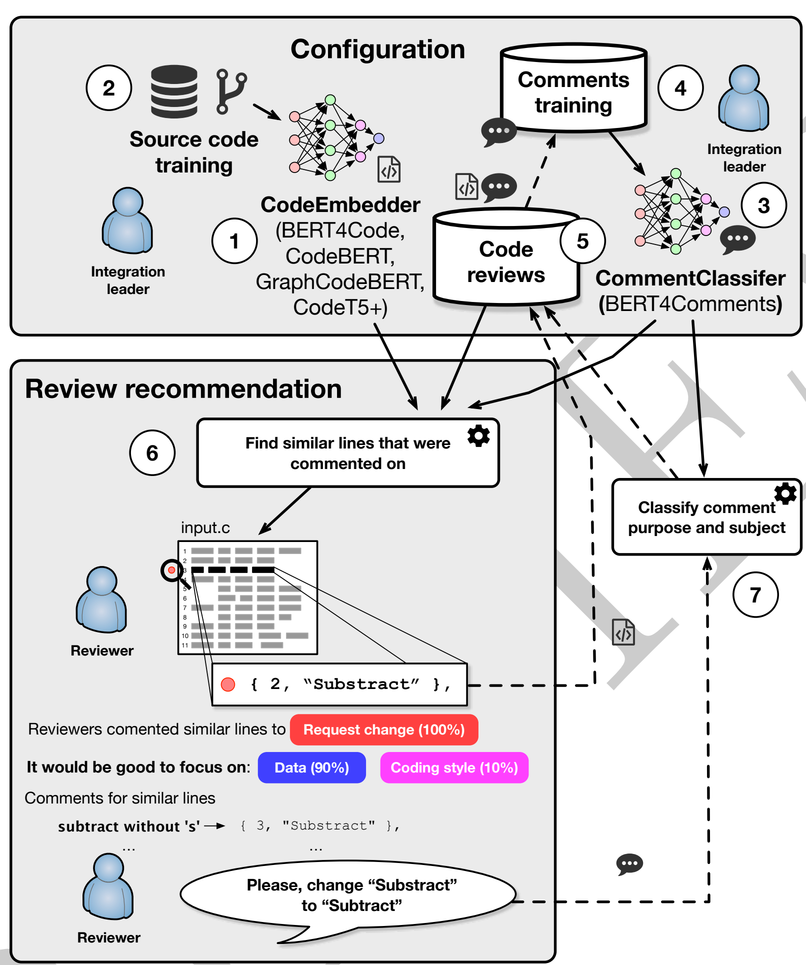

Figure 1. Overview of the use of ACoRA. First, the integration leader configures ACoRA. Then, software developers use ACoRA to check their code before it is integrated with the entire product code.

The integration leader, or another continuous integration specialist, configures ACoRA once when the system is being set up. Later, this process can be periodically repeated in order to update the recommendations. Software developers and architects use ACoRA to check the quality of their source code before it is integrated with the entire codebase of the product. Since ACoRA uses code fragments, it can be used as part of the continuous integration toolchain or as an add-on to a development environment.

The configuration process presented in Figure 1 consists of two independent subprocesses. The first one is to pre-train/fine-tune a neural network language model ① (CodeEmbedder) using a codebase ② that seems similar to the target codebase on which one wants to apply ACoRA. Alternatively, one can use publicly available pre-trained models (e.g., CodeBERT,

[2] ACoRA — https://github.com/mochodek/acora

PREVIEW

CodeT5+, etc.). The second sub-process is to train/fine-tune a review-comment classifier ③ (CommentClassifier) using a dataset of manually labeled comments ④. Finally, one needs to establish a reference database of past code reviews ⑤ that includes commented code chunks and reviewer comments classified with CommentClassifier.

A configured ACoRA can provide recommendations ⑥ as shown in Figure 2 by searching for similar lines in the reference code reviews database ⑤ to those currently under review. As we can see in Figure 2, ACoRA suggested the reviewer to focus on line 3 by providing recommendations about the potential nature of the problem (code_data) and an example of a similar line from the reference code database. New comments provided by reviewers can be classified with CommentClassifier and stored in the database ⑤.

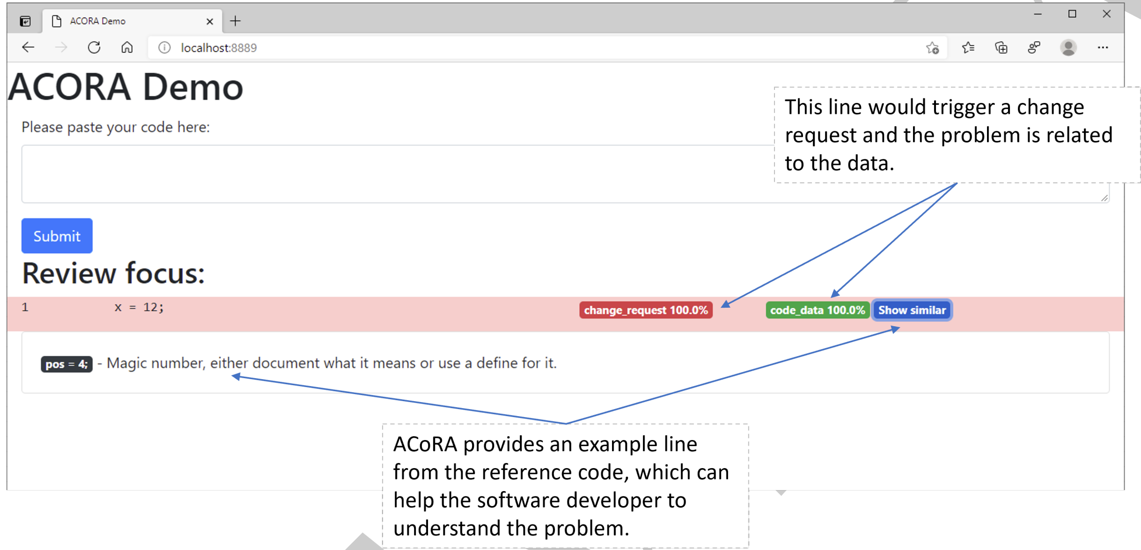

Figure 2. Demonstration of ACoRA as a stand-alone web service, to be used by software developers. After submitting a code fragment, different lines in this fragment are provided with recommendations on what to fix.

3.1. CodeEmbedder Language Models

The core component of ACoRA is a language model used to transform programming language text to its vector representation. As the ACoRA design follows the pipe and filter architectural style, one can either use a built-in infrastructure to train such a model (BERT4Code) or employ one of the publicly available pre-trained models for generating code embedding vectors by adding a new filter component. For instance, in this study, we use three proven pre-trained models, i.e., CodeBERT, GraphCodeBERT, and CodeT5+, as well, as BERT4Code models pre-trained from scratch.

The BERT4Code model is based on the BERT (Bidirectional Encoder Representations from Transformers) language model [20], which is a deep artificial neural network implementing a multi-layer bidirectional Transformer architecture [15] (however, technically, it uses only the Transformer Encoder stack). The language model is trained and evaluated during the configuration phase of ACoRA and used in the recommendation phase. The pipeline for training follows the same principles as the established approaches like TransCoder and CodeGen [40]:

PREVIEW

1. Tokenization: where each code fragment is transformed into a set of tokens. We use a modified version of WordPiece tokenizer [41] that split tokens not only based on whitespace characters but also on operators, brackets, etc. [42]. Optionally, `ACoRA` allows to convert the out-of-vocabulary tokens into their symbolic form [43], e.g., a variable identifier `number0` would be replaced by a signature `a0` (a sequence of small letters proceeded with a sequence of digits).
2. Padding: where each code fragment is transformed to a vector of the same size, 128 tokens in our case.
3. Embeddings extraction: a fragment of code is transformed into its embedding representation using four last hidden layers of `BERT4Code` (by following the recommendations for the original `BERT` model [20]).

In the training phase, we pre-train a `BERT4Code` model using the same procedure as for the original `BERT` model [20]. This process is used by other BERT-inspired models [44]. The inputs to the model are pairs of code fragments (in 50% of cases these are consecutive fragments). The model is simultaneously trained on two tasks—Masked Language Model (MLM) and Next Sentence Prediction (NSP). For the former, 15% of tokens in a sequence is being masked and the goal of the network is to guess the original ones. For the NSP task, the network needs to respond to whether the second provided code fragment directly proceeds from the first one. The `BERT4Code` models studied in this paper consist of four layers (384 neurons in each of the hidden layers; 8 attention heads). We use compact `BERT` models [45] since programming languages are more formal (and structured) than natural languages. Also, such networks can be pre-trained on commodity hardware affordable even for small organizations.

In the inference phase, each code fragment is inputted to the `BERT4Code` model, and the embedding vector output is used when calculating the similarity between code fragments.

3.2. `CommentClassifier` Language Model

`ACoRA` provides a default implementation of `CommentClassifier` called `BERT4Comments`[3]. It is a language mode structurally similar to the `BERT4Code` model, except that it is based on the official pre-trained `BERT` model (12-layer), which is further fine-tuned to classify review comments. The `BERT` model was pre-trained on a large corpus of plain text for the masked word prediction and next sentence prediction tasks. Such a pre-trained `BERT` model can be further fine-tuned to a specific downstream task. The input to `BERT4Comments` is one review comment and the output is a set of categories describing what the comment is about. The process is shown in Figure 3.

The taxonomy of the comments is taken from our previous work [16], and is shown in the top row of Figure 3. The categories of the taxonomy are as follows:
code_design — the comment is about a structural organization of code into modules, functions, classes, or similar, e.g. “code snippet inherited from original dissector. I have refactored the code to have the decompression in a single place now it should be a bit better”. It is also about overriding, e.g. “this will not work for IA5. Why not simply override dataCoding before the switch?”, and dead/unused code, e.g. “This is duplicated code, put outside the if-else.”.

[3]BERT4Comments is also available at the Huggingface repository: https://huggingface.co/mochodek/comment-bert-subject together with a complementary model to annotate comment purpose https://huggingface.co/mochodek/comment-bert-purpose.

PREVIEW

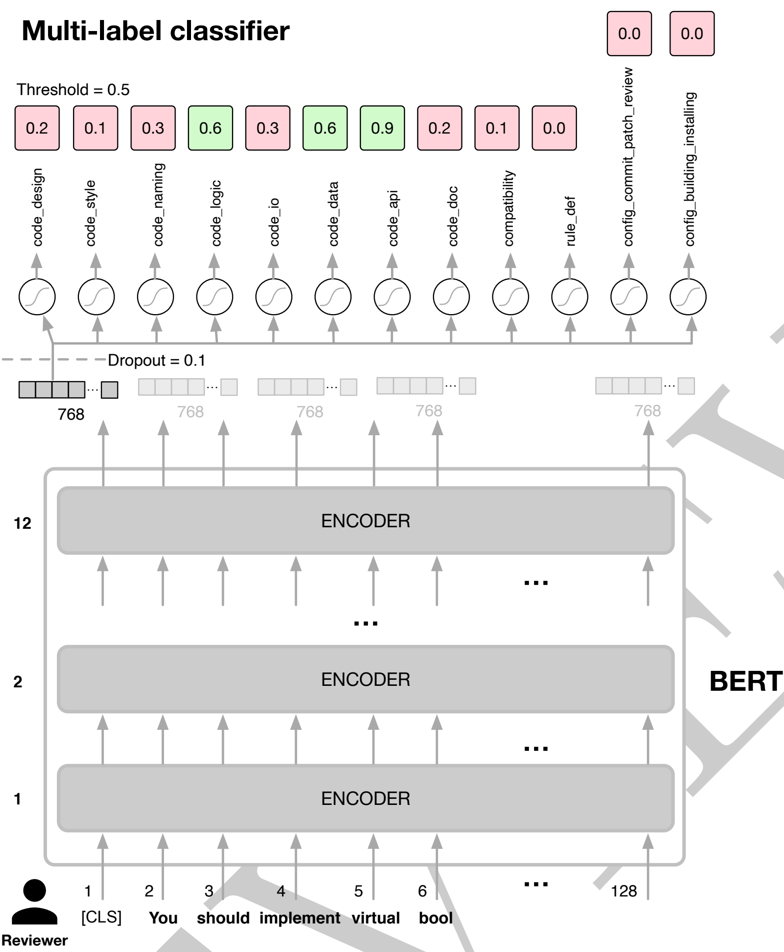

Figure 3. `BERT4Comments` architecture [16].

PREVIEW

code_style — the comment is about the layout of the code/readability issues, for example: "add blank line" or "formatting: remove space after 4".
code_naming — the comment is about issues related to naming code constructs, tables, for example "please use lowercase for field name => 'isakmp.sak.nextpayload' " or "Add name of dissector XXX: use custom..."
code_logic — the comment is about algorithms used, operations on data, calling functions, creating objects, and also the order the operations are performed, for example "missing validation of chunk size, potential buffer overflow?" or "should this be initialized with NULL or something?"
code_io — the comment is about input/output, GUI, for example: "What about showing the hub port, i.e. 'address:port'? So the normal endpoints would display as 'address.endpoint' and split would display as 'address:port' " or "Debug output to be removed?".
code_data — the comment is about data, variables, tables, pieces of information, and strings, for example: "You probably want encoding ENC_BIG_ENDIAN here. You could also use
proto_tre_add_item-ret()int() here to avoid fetching the value twice. This is true for other places in the code too" or "Are these ports registered with IANA? If not, I am not sure if they should be used here".

code_api — the comment is about an existing API or suggestions on how the API should evolve, for example: "This needs to remain the same as before. The dissection must continue therefore the latest offset must be updated after adding to the tree. offset += dissect_dsmcc_un_session_nsap(tvb, offset, pinfo, sub_sub_tree)" or "If they are non-standard and uncommon, I would replace them with: dissector_add_for_decode_as("udp.port", otrxd_handle)".
code_doc — the comment concerns the documentation or comments in the source code, for example: "Which 3GPP document specifies this AVP?" or "Maybe remove this comment now? We do not support older drafts anymore".
compatibility — the comment is related to the operating system compatibility, tools compatibility, versions, or issues that appear only on certain platforms, e.g.: "the Ubuntu failure is to the revert of my previous change (only on_btnImport
_clicked() call must be guarded)" or "I can empirically confirm that /proc/self/exe somehow expands to the real path. So this code would probably have no effect on Linux".
rule_def — the comment can be used to elicit a definition of coding/style rules, note it has to explain the broader context, e.g., "'add blank line' is not a definition since we don't know why the blank line should be added here; on contrary, 'use space for indent (like rest of file)' states that spaces should be used for indentation (in general)" or "remove comment when it doesn't help understanding the code".
config_commit_patch_review — the comment is about patches, commits, or review comments, for example: "To be done in the next patch set" or "Right, if you decide to do a formatting patch, it is best to do that in a separate change".
config_building_installing — the comment is about a process of building, installing, and running the product, for example "This is not required. Already done by the install script" or "let's remove this example, installing binary packages across different distros is not supported and we should not recommend users to skip signature checking, etc".

Naturally, this taxonomy can be used manually to understand and classify each comment, but the best support is to use an automated classifier of these comments. The classifier needs to be based on techniques from natural language processing and has to utilize a pre-trained model as the number of comments in a typical repository is not in parity with the diversity of the natural language constructs available.

The proposed taxonomy consists of categories representing high-level problem areas to direct the focus of reviewers. However, it is possible to extend or even replace our taxonomy with other taxonomies like the one proposed by Tufano et al. [34] that defines low-level code issues that are raised during MCRs.

In our previous study [16], we trained and validated `BERT4Comments` models on the dataset of 2,672 MCR comments from three open-source projects, i.e., Wireshark, The Mono Framework, and Open Network Automation Platform (ONAP). The accuracy of the models ranged from 0.84 to 0.99 (mean = 0.94). The mean Matthews Correlation Coefficient (MCC) value was equal to 0.60, which can be considered a moderate to strong correlation [46]. Finally, the average Area under the ROC Curve (AUC) was equal to 0.76, which is an acceptable value for a classifier according to [47].

PREVIEW

4. Research Methodology

In this work, we use design science research (DSR) methodology as described by Wieringa [17]. The first step recommended by DSR is problem formulation, which usually entails the need to organize artifact development and treatment design into more distinct parts [48].

The evaluation is structured into the initial evaluation, where we study open-source projects in-depth, and the external evaluation, where we apply `ACoRA` on an industrial project with an industrial partner—Bosch Gmbh.

The reproduction package for the study containing datasets and scripts used to perform the analyses (with the exclusion of confidential data) is publicly available.[4]

4.1. Problem formulation

In the context of our study, the question of *"To what degree can we suggest relevant review guidance for a given code fragment based on historical data?"* has two parts, two sub-questions:

RQ1: To what degree does the language embedding model find similar code fragments?
RQ2: How relevant are the review comment suggestions with respect to the nature of the problem identified by reviewers?

RQ1 addresses our need to understand how well language embedding models (`CodeEmbedders`) find similar code fragments. We need to know whether the lines that are matched as similar are relevant—whether two given lines can be judged as similar or not. The similarity is a concept that depends on the textual content of the line, its context, and the semantics of a line, and there is no good measure for that [49]. Therefore, in our work we approximate similarity by changes in code fragments—we can change a code fragment so that it is textually different but has a resemblance in terms of its purpose and context. We seed changes to code fragments and assess the fragments returned by the language model as similar. The changes are designed to exemplify types of problems that the reviewers comment on, i.e., the taxonomy presented in Section 3.2 and studied in [16].

RQ2 addresses the problem of how well the review suggestions correspond to the suggestions that a code reviewer would provide. Since the review comments are specific to the code fragments commented on, we need to generalize them, and therefore we use the comment taxonomy and `BERT4Comments` as the means to provide such suggestions. As learning-by-example has been identified as an important aspect of MCR, we provide a similar line and its comment as an example. **RQ2** also addresses the challenge related to the reproducibility of suggestions and their generalizability. Since the review comments are specific to the code fragment, they cannot be directly provided as suggestions for other code fragments (even similar ones). However, they can capture problems that can be generalized and these generalized suggestions can be used as guidance for the software developers reviewing the new code fragment.

[4] Reproduction package — https://www.cs.put.poznan.pl/mochodek/acora-package.7z. *Note: this is a temporal location for the purpose of the manuscript-review process.*

PREVIEW

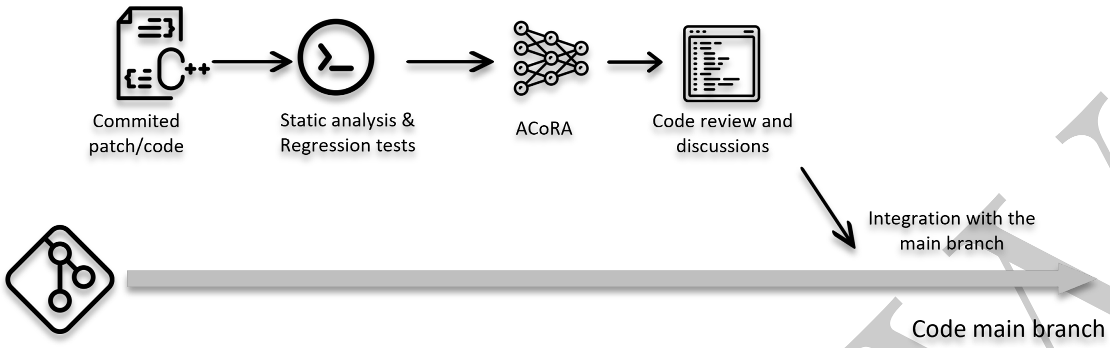

Figure 4. `ACoRA` integrated into a CI pipeline.

4.2. Treatment design

To address the research questions and ensure that `ACoRA` supports both the automation of code reviews and the possibility of training new project members, it is integrated into a continuous integration pipeline, as shown in Figure 4.

The integration is done using docker containers, i.e., we designed `ACoRA` as a microservice that can be plugged into a continuous integration pipeline. It has several components designed according to the pipes and filters architectural style that allow for flexibility in implementing and replacing the components. The tool can be configured to provide feedback offline, as shown in Figure 2. The detailed design is described in Section 3.

4.3. Initial treatment evaluation and improvements

To evaluate the code review platform, we chose to use two separate open-source projects, which had an open Gerrit code review tool instance available for public access. For the initial treatment evaluation, we chose two open-source projects:

- Wireshark (https://www.wireshark.org) – an open-source network protocol analyzer, developed by professionals from leading telecommunication companies.
- Cloudera (https://www.cloudera.com) – an open-source cloud hosting solution, developed by a professional community.

Both projects are professionally developed and we chose them because of the size and quality of the code reviews in Gerrit. These two products are also rather specific, so the variety of use cases is not considered as a factor in assessing code reviews. In particular, when discussing the selection with our industrial partner, we wanted tools that were either in C/C++ or Java, developed by professional developers, and had a specific purpose. We considered using other tools like Android to study, but the diversity of the product (mix of C, C++, and Java), as well as the number of use cases (it is an operating system), made it not fit our requirements.

First, we prepared for the evaluation by performing the following steps:

Step 1: Clone a repository at a selected revision (#base_rev).

PREVIEW

Step 2: Obtain a `CodeEmbedder` model either by pre-training `BERT4Code` on the code downloaded in Step 1 or by choosing one of the pre-trained language models, i.e., `CodeBERT` [21], `GraphCodeBERT` [50], or `CodeT5+` [51].
Step 3: Fetch the MCR review comments for the revisions following the #base_rev and the specific code fragments that were reviewed (*all-code dataset*).
Step 4: Use the `CommentClassifier` (i.e., `BERT4Comments` trained on the manually labeled dataset of MCR comments [16]) to classify the MCR comments in the *all-code dataset*.

Then, we performed the following steps to address **RQ1**:

Step 5: Select a sample of the code fragments from the *all-code dataset* (up to 11 fragments per each of the taxonomy categories).
Step 6: Introduce changes to the code fragments based on the taxonomy (*modified dataset*).
Step 7: Assess the similarity of the matched code fragments.

We decided to select and study four Transformer-based `CodeEmbedders` (Step 2). First, we pre-trained `BERT4Code` according to the process described in Section 3.1. This represents a scenario where a `CodeEmbedder` is pre-trained on a small but very representative codebase (an intra-organization usage). Then, we employed three language models pre-trained on large code corpora that are publicly available, i.e., `CodeBERT`, `GraphCodeBERT`, and `CodeT5+`. The former model is a BERT-based model pre-trained on combined inputs of natural language texts and code. `GraphCodeBERT` augments the inputs with the information about data flow. Finally, `CodeT5+` is an example of a large Transformer-based model. Although our goal was not to determine what is the best possible `CodeEmbedder`, the variety of selected models allowed us to gain early insights regarding the differences in how `ACoRA` performs depending on the embedding model it uses.

In Step 6, we modified lines to investigate how sensitive to such changes `CodeEmbedders`' embeddings are when they are used to find similar lines. We selected up to 10 lines for each of the comment categories from our taxonomy and introduced the following types of changes related to the category of comments they belong to:

– **config_commit_patch_review** — since most of the comments belonging to that category refer to lines of code that are commented, we decided to add/modify/replace particular parts of the text in a line, e.g., a person's name, commit identifier referred to in a line, a part of variable identifier, etc.,

> Examples:
> Original: ❝`* Copyright (C) 2010-2012 The Async HBase Authors. All rights reserved.`❞
> Modified: ❝`* Copyright (C) 2010-2012 The another company Authors. All rights reserved.`❞
> Original: ❝` * scanners through the @link KuduScanToken API. Or use MapReduce.`❞
> Modified: ❝` * scanners through the @link AnotherPatch API. Or use MapReduce.`❞
> Original: ❝`Change-Id: I1d6dfc4314091eb6f3eef418c5a17ed37f7a1200`❞
> Modified: ❝`Change-Id: I2d6dfc4314091eb3d3eef418c5a17ed37f7a1200`❞

– **code_logic** — we changed the logic of the code by modifying the operators being used,

PREVIEW

```
Examples:
Original: ❝ return percent/4;❞
Modified: ❝ return percent*4;❞
Original: ❝  else if (containers_total_ != 0) ❞
Modified: ❝  else if (containers_total_ == 0) ❞
Original: ❝ offset += 4_❞
Modified: ❝ offset -= 4_❞
```

– **code_data** — we changed either the values or the types of variables,

```
Examples:
Original: ❝ int64_t time_elapsed = 0;❞
Modified: ❝ int32_t time_elapsed = 0;❞
Original: ❝ percent += 100;❞
Modified: ❝ percent += 1000;❞
```

– **code_style** — we added/removed indentation/trailing whitespaces,

```
Examples:
Original: ❝        int num_micro_batches = ❞
Modified: ❝int num_micro_batches = ❞
Original: ❝    if (packet_num > 0) {❞
Modified: ❝        if (packet_num > 0) {❞
```

– **code_doc** — we commented/uncommented lines using inline or block comments,

```
Examples:
Original: ❝ return 11_❞
Modified: ❝ // return 11_❞
Original: ❝/* packet_list.cpp❞
Modified: ❝packet_list.cpp❞
```

– **code_io** — we modified sub-words in identifiers to their synonyms or changed abbreviations to full words,

```
Examples:
Original: ❝ LOG(ERROR) << s;❞
Modified: ❝ LOG(PROBLEM) << s;❞
Original: ❝ ctx_menu_.addSeparator()_❞
Modified: ❝ context_menu_.addSeparator()_❞
```

– **code_api** — we added/removed some parameters from function headers/calls,

```
Examples:
Original: ❝ ascendlex_destroy(&scanner)_❞
Modified: ❝ ascendlex_destroy(&scanner, force)_❞
Original: ❝ DCHECK_EQ(out_length, cleartext[i].size());❞
Modified: ❝ DCHECK_EQ(out_length, cleartext[i].size(out_length));❞
```

– **code_naming** — we changed the character case for some identifiers,

PREVIEW

> **Examples:**
> Original: “`#define USBLL_POISON 0xFE`”
> Modified: “`#define usbll_poison 0xFE`”
> Original: “ `gint bytes_consumed_`”
> Modified: “ `gint Bytes_Consumed_`”

– **code_design** — we modified/removed the keywords or names of identifiers,

> **Examples:**
> Original: “ `if ( skipped > 0 )`”
> Modified: “ `while( skipped > 0 )`”
> Original: “ `LZ4F_freeDecompressionContext(lz4_ctxt)_`”
> Modified: “ `LZ4F_freeCompressionContext(lz4_ctxt)_`”

– **compatibility** — we added a suffix `_v2` to some identifiers,

> **Examples:**
> Original: “ `struct hf_tree tree = 0_`”
> Modified: “ `struct hf_tree tree_v2 = 0_`”
> Original: “`find_path(PCAP_INCLUDE_DIR`”
> Modified: “`find_path_v2(PCAP_INCLUDE_DIR`”

To assess the matched code fragments in Step 7, we used the ranking of the recommendations. The fragment that is returned as the closest one (calculated as the Minkowski distance, used by `ACoRA`, between the embedding vectors) is ranked 1, the second closest is ranked 2, and so on. For practical reasons, we only report up to rank 10, and for higher ranks, we only report that the rank is higher than 10. In this case, we assess how well `CodeEmbedder` can match the modified line to the original line.

To address **RQ2**, we focused on evaluating the quality of `ACoRA`'s recommendations by performing the following steps:

Step 8: Divide the *all-code dataset* into *reference database* and *validation dataset* based on a selected revision (#split_rev)—remove duplicates so both datasets consist only of unique code fragments.
Step 9: Use `ACoRA` to provide recommendations for the code fragments in the *validation dataset* based on the *reference database*.
Step 10: Assess the relevance of the recommendations.

To evaluate the relevance of the suggestions, we start by dividing the code fragments in *all-code dataset* into the *reference database* and *validation dataset* to mimic how `ACoRA` would be used. We eliminate the duplicated code fragments so we do not bias the results by over-representing a given code fragment. We then use the *reference database* as a basis for providing recommendations for the code fragments in *validation dataset*. We apply the following filtering criteria to mitigate the impact of the following issues related to data quality:

– MCR comments attached to empty lines or code-block opening/closing — we remove lines that contain less than three characters—unfortunately, we observed that sometimes comments concerning fragments of code are attached by reviewers to lines such as closing or opening of code blocks or to empty lines proceeding or preceding a given code fragment. As a result, there is no logical association between the comment provided by the reviewer and the line of code in the data.

PREVIEW

- Acknowledgment comments — we remove comments being classified by the `BERT4Comments` model as `acknowledgment` (e.g., "Thank you", "Done") since these are irrelevant from the perspective of providing suggestions to reviewers.
- Reviewers' discussion not resulting in change requests — we remove the lines for which there were no comments classified by `BERT4Comments` as `change_request`. Since not all comments provided by reviewers have to be relevant, we assume that the discussions between reviewers that do not result in requesting a change to be made in code are not useful as the basis for automatic recommendations.

In order to evaluate the correctness of the suggestions, we calculate how many of the suggested categories overlap with the original categories, compared to the number of categories in the original line, according to Formula 1. In the formula, O is the overlap ratio, N is a set of categories of comments in the new line and R is the set of categories of comments for the reference line.

$$O = \frac{|N \cap R|}{|N|} \tag{1}$$

Formula 1 is designed so that the correct suggestion means that the overlap ratio is 1.0 while a suggestion that misses all categories is 0.0. A partial suggestion is between these values, with the better suggestions being closer to 1.0.

Finally, the quality of recommendations provided by automatic tools such as `ACoRA` depends on how similar the lines and comments in the *reference database* are with respect to the new code being targeted for review. Therefore, we perform the analyses for three scenarios. In the first scenario, we force `ACoRA` to provide recommendations for every line in *validation dataset*, no matter how similar or different are the reference and validation lines. In the following two scenarios, we use a maximum distance threshold between the line-embedding vectors generated by `CodeEmbedder`s to control whether a recommendation should be made or not. We calculate these thresholds based on the distribution of distances between the most similar lines in the *reference database* (10th and 50th percentile). We assume that lowering the distance threshold should result in increasing the relevance of comments with the cost of decreasing the number of cases for which the recommendation could be provided.

4.3.1. Wireshark and Cloudera

In the Wireshark and Cloudera projects, we focused on the subset of the projects written in C as our goal was to use the same base language model for all three evaluations. In the industrial validation, we limited the search to a smaller number as our visit time was limited. Project-specific parameters are presented in Table 1.

These three projects are selected since we validate slightly different aspects in each context, as prescribed by both design science research [17] and action research [48]. In Wireshark, we focused on the curation of the dataset for training. In Cloudera, we focused on the scalability of the approach and sensitivity to noise in the dataset, and in industry, we focused on the usability of the suggestions from the perspective of a software developer.

The `BERT4Comments` used in the study was trained on a subset of manually curated and labeled Wireshark codebase. The curation was done by:

PREVIEW

Table 1. Project specific parameters used in our research design. *Note 1: The Wireshark community has migrated from Gerrit to Gitlab and its old Gerrit instance is no longer available therefore we only use data up until 2019-12-31. Note 2: We are not allowed to publish some information regarding the industrial source code (N/A).*

Parameter	Wireshark	Cloudera	Industry
Revision (#base_rev) (Step 1)	5e34492a7e	10e3cec127	N/A
Pre-train `BERT4Code` (Step 2)	20 epochs, batch size 32, sequence length 128	the same	the same
Review comments (Step 3)	2015-02-23	2022-01-26	2021-11-24
Reference database (Step 8) – comments – unique code fragments	 40,430 9,930	 74,000 29,599	 N/A 15,000
Validation dataset (Step 8) – comments – unique code fragments	 46,850 12,789	 174,630 51,034	 N/A 10 commits
Filtered suggestions (Step 9)	1,582	4,738	10 commits

- Removing the pairs `<line, comment>` which are wrong, e.g., when the review comment does not comment on anything specific to that line.
- Rewriting the comment to make the text more general and less specific to a particular line, e.g., by changing a comment about datatype `int` to a comment about a datatype in general.

We used the same approach in our previous work and the accuracy and MCC for `BERT4Comments` were 0.86–0.98 (Accuracy) and 0.32–0.62 (MCC) [16]. Therefore, the evaluation of `BERT4Comments` is outside of the scope of this article.

4.4. Evaluation at the industrial partner

To understand the limitations of `ACoRA`, we design an evaluation of the tool at our industrial partner. The evaluation was done in the following way.

First, we pre-trained `ACoRA`'s `BERT4Code` on the source code from an open-source project suggested by the company employees as being similar to their code (19,218,513 lines of code). Next, we fetched code and comments from a Gerrit instance of a selected project at the company. For practical reasons, we limited the dataset to 500 patches, which included approximately 18,000 lines of code—15,000 with comments and 3,000 without comments. The commented lines were used as the *reference database* in this study.

Second, we tested `ACoRA` on 10 selected commits from another project as the *validation dataset*. We asked the company representatives to extract 10 commits containing code fragments that were commented on by a code reviewer, based on the method used to evaluate software measures [52]. We used `ACoRA` to provide suggestions for the code based on the comments in the *reference database*. The company representatives were asked to judge the correctness and actionability of each recommendation. We adjusted `ACoRA` to be over-sensitive, i.e. provide more suggestions for reviews, in order to challenge the industrial partners to check whether more lines should be commented on in the evaluated commits.

PREVIEW

5. Results

5.1. Wireshark

5.1.1. RQ1: Finding similar lines – Wireshark

The recommendations provided by ACoRA are based on finding similarities between the lines under review and the lines previously commented on by reviewers. ACoRA performs that task by measuring the distance between line embeddings generated by CodeEmbedder. Unfortunately, the neural-network-based language models work as black boxes and we cannot tell exactly what relationships between tokens and code lines they capture. Therefore, we decided to study this phenomenon by investigating how sensitive the selected CodeEmbedders' code-line representations are to certain types of modifications introduced to lines when used to search for line similarities. We modify a line of code, ask ACoRA to search for similar lines to the modified one, and calculate the ranking position of the original line in the lines recommended by ACoRA. If the ranking position of the original line is greater than one, it means that the change introduced to the line caused it to be more "similar" to some other lines in the dataset than to the original line from which it was derived. The dataset included 35,000 Wireshark lines (including the 100 original lines that were modified), therefore, it was rich in examples of lines that could be identified as more similar to the modified lines than the original ones.

Figure 5 shows the distributions of ranking positions of original lines in the recommendations provided for their modified counterparts in the dataset when CodeT5+ was used as a CodeEmbedder. Although none of the CodeEmbedders appeared unanimously superior, ACoRA using CodeT5+ seems to provide the best overall results (the results for the remaining CodeEmbedders are presented in Table 2). For all CodeEmbedders, the changes made in the commented lines belonging to config_commit_patch_review and code_style categories appeared as difficult. Also, BERT4Code, CodeBERT, and GraphCodeBERT all had problems with detecting similarities for the code_doc category, which was not the case for CodeT5+. Finally, CodeT5+ performed slightly worse than BERT4Code for the code_data category.

We analyzed each of the cases where the original line was not provided as the first recommendation to study and hypothesize about what differences in code make the studied CodeEmbedders' embeddings recognize lines as similar or not.

For the changes made to the lines belonging tocode_io, the top suggestions made by ACoRA were the original lines. Therefore, the changes made to these lines were not significant enough (with regards to how the lines are represented by CodeEmbedders' embeddings) to make any other lines more similar to them than their original lines.

config_commit_patch_review — ACoRA using CodeT5+ found five perfect matches (the original line appears as the first recommendation). These lines were modified by changing the e-mail addresses of reviewers, changing the commit hash, or a website address. CodeBERT and GraphCodeBERT failed to match the lines with modified e-mail addresses, while BERT4Code matched only the lines with modified change hashes. Still, the recommendations for the remaining lines in this category seemed valid, despite the fact that the original lines were not provided as top recommendations. For instance, when we modified one of the lines having the structure of Petri-Dish: Name Surname <e-mail

PREVIEW

Table 2. Similarity of modified lines to their original counterparts for Wireshark (all `CodeEmbedder`s).

`CodeEmbedder` / taxonomy category	**original line similarity rank**										
config_commit_patch_review	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	2	1	2	-	-	1	-	-	-	-	4
`CodeBERT`	5	-	-	-	1	-	-	1	-	-	3
`GraphCodeBERT`	4	-	-	-	-	1	-	-	-	-	5
`CodeT5+`	5	1	1	-	-	-	-	-	1	-	2
code_logic	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	8	1	-	1	-	-	-	-	-	-	-
`CodeBERT`	9	-	-	1	-	-	-	-	-	-	-
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	9	1	-	-	-	-	-	-	-	-	-
code_data	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	10	-	-	-	-	-	-	-	-	-	-
`CodeBERT`	9	-	-	-	-	-	-	-	-	-	1
`GraphCodeBERT`	9	-	-	-	-	-	-	-	-	-	1
`CodeT5+`	9	-	1	-	-	-	-	-	-	-	-
code_style	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	1	-	-	-	-	-	-	-	-	-	9
`CodeBERT`	4	-	-	-	1	-	-	-	-	-	5
`GraphCodeBERT`	8	-	-	-	1	-	-	-	-	-	1
`CodeT5+`	9	-	-	-	-	-	-	-	-	-	1
code_doc	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	1	-	2	-	-	-	-	-	-	-	8
`CodeBERT`	6	-	-	-	1	-	-	-	-	-	3
`GraphCodeBERT`	8	-	-	-	-	-	-	-	-	-	2
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-
code_io	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	10	-	-	-	-	-	-	-	-	-	-
`CodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-
code_api	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	10	-	-	-	-	-	-	-	-	-	-
`CodeBERT`	8	-	-	-	-	-	-	-	-	1	1
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-
code_naming	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	4	3	-	1	-	-	-	-	-	-	2
`CodeBERT`	8	-	1	-	-	-	-	-	-	-	1
`GraphCodeBERT`	9	-	-	-	-	-	-	-	-	-	1
`CodeT5+`	9	1	-	-	-	-	-	-	-	-	-
code_design	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	6	-	-	-	-	-	-	1	-	-	3
`CodeBERT`	9	-	-	-	-	-	-	-	-	-	1
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-
compatibility	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	10	-	-	-	-	-	-	-	-	-	-
`CodeBERT`	9	-	-	-	-	-	-	-	-	-	1
`GraphCodeBERT`	9	-	-	-	-	-	-	-	-	-	1
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-

PREVIEW

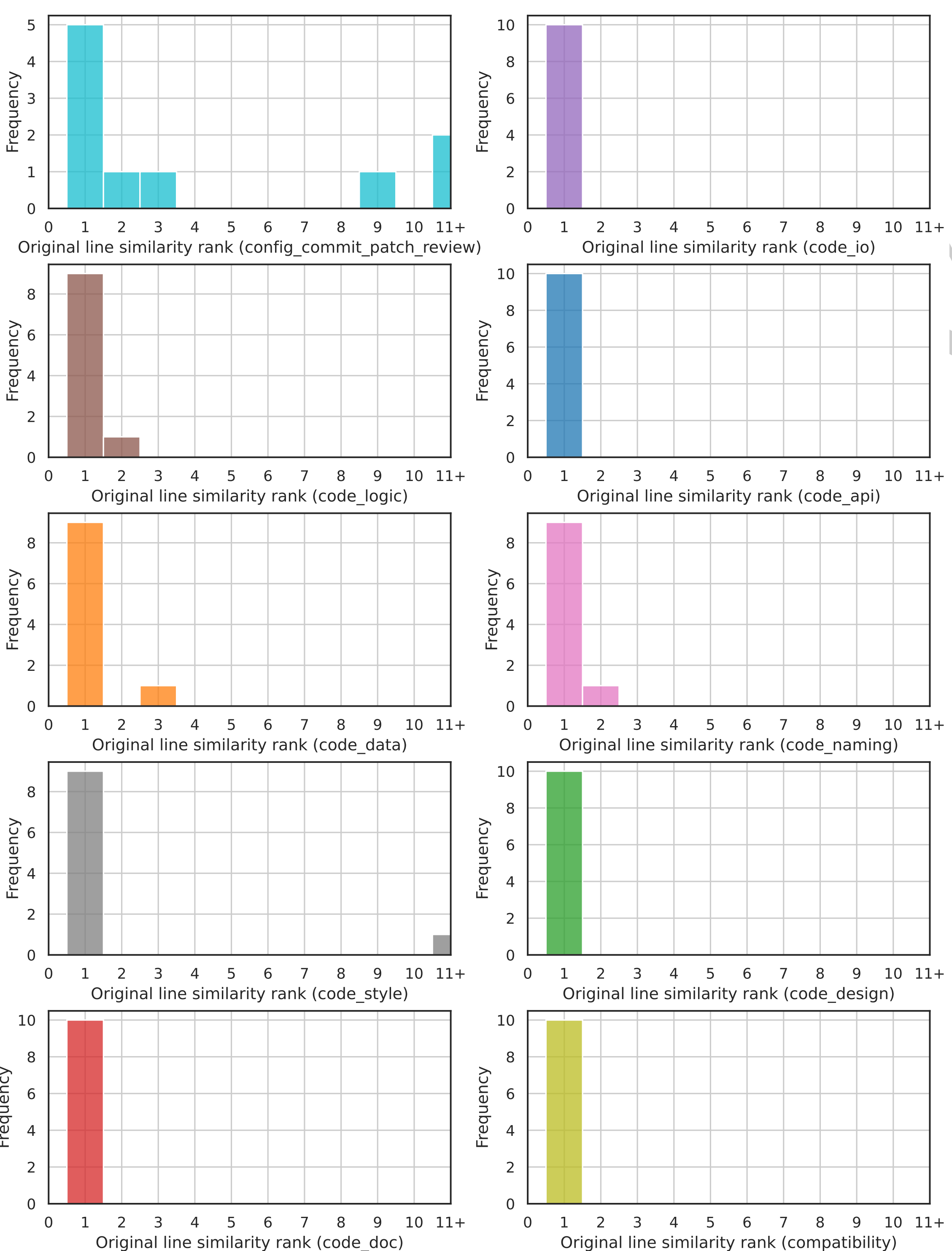

Figure 5. Similarity of modified lines to their original counterparts for Wireshark (`CodeT5+`).

`address>` by changing the person's name and e-mail address, the original line appeared as the sixth recommendation, however, all `ACoRA` suggestions seemed equally valid since they all had exactly the same structure but contained names and addresses of other people. We made similar observations for other lines, e.g., for the line "`Reviewed-on: https://code.wireshark.org/review/33705`" that we modified by changing the review identifier to `52705`. The original line appeared as the recommendation number 42, however, all proceeding lines in the ranking had exactly the same structure but different review identifiers (e.g., "`Reviewed-on: https://code.wireshark.org/review/22515`").

`code_logic` — the changes we made for commented lines belonging to this category were mostly simple operator overloading (e.g., "<" → ">"). `ACoRA` using `CodeT5+` was able to indicate the modified line in the first place in nine out of ten cases. However, in the only case that the modified line was ranked second, the indicated similar line seemed more similar to the original line than its modified version. The original line was " `*buf = '\0'_ /* NULL terminate */`" and was modified by removing the pointer operator `*`. The most similar line found by `ACoRA` was " `*buf = '\0'; /* NULL terminate */`" which differs by a single character and has the pointer operator. Interestingly, `ACoRA` using `GraphCodeBERT` indicated all modified lines as the most similar to the original ones. However, taking into account the above, this might not necessarily be the best result. Finally, `BERT4Code` performed slightly worse and suggested eight out of ten modified lines. The two exceptions were the lines: "`if (tree) {`" modified to "`if ( ! tree) {`" and "`DISSECTOR_ASSERT(pos+chunk_size <= length)_`" changed to "`DISSECTOR_ASSERT(pos+chunk_size > length)_`". In the case of the former, the line that was considered more similar by `ACoRA` was the line "`if (len > length) {`"—a conditional expression that has the same number of indenting whitespace characters but uses a different operator (">" instead of negation). The original line has no operator at all, which might be the cause of recognizing this line as more similar. For the latter line, the recommendations were less clear to us. The top recommended line was "`g_free(temp_name)`", which is also a function call, but in contrast to the modified line, there are no operators used to calculate the values of function parameters being passed.

`code_data` — only `ACoRA` using `BERT4Code` provided all ten modified lines as top suggestions, however, the remaining `CodeEmbedder`s were also very accurate with nine out of ten correct suggestions.

`code_style` — we observed mixed results for this category. `ACoRA` using `CodeT5+` or `GraphCodeBERT` correctly indicated nearly all modified lines (nine and eight, respectively), while for `BERT4Code` and `CodeBERT` in nine and five cases the modified line was ranked beyond top ten suggestions. After analyzing the worst cases, we could observe that tab (⎵) was often "recognized" as a visibly different token than space (␣). For instance, when we removed indentation spaces from the line "`␣␣␣␣if (api_version >= 4 ) {`", the most similar line suggested by `ACoRA` was "`⎵if (tokenlen >= 1) {`". Similarly, the top match for the line "`⎵⎵if(is_encrypted && !docsis_didssect_en crypted_frames)`" modified by changing two indentation tabs (⎵⎵) to four spaces (␣␣␣␣) was "`␣␣␣␣if (is_from_server && session->is_session_resumed)`". Therefore, the presence of indentation spaces influenced, to a large degree, the embedding vector for these lines. These observations were consistent among the other studied examples. Therefore, it seems that using `BERT4Code` or `CodeBERT` embeddings for finding similar lines is sensitive to the number and type of indentation whitespaces. We hypothesize that it might be a consequence of how `BERT` models are trained. One of the tasks used to pre-train

PREVIEW

BERT is next sentence prediction (i.e., next line of code prediction in the case of BERT4Code). In programming languages, such as C/C++, whitespaces are used by programmers to indent code blocks to improve code readability. Therefore, it seems that the number of trailing whitespaces could be an important code-feature that a BERT model uses when it is pre-trained on source lines of code. Also, using spaces for indentation (typically 2 or 4) is preferred over tabs within the Wireshark community, therefore, using tabs for that purpose could be considered as an anomaly. Finally, this property of BERT4Code does not have to be necessarily perceived as its weakness, since it could help identify issues related to wrong indentation in the code.

code_doc — again, we observed that ACoRA using CodeT5+ or GraphCodeBERT was visibly better in finding modified lines than the variants using BERT4Code or CodeBERT. CodeT5+-based ACoRA correctly indicated all of the modified lines and provided them as top-ranked suggestions. At the same time, it seems that the toggle between a commented/non-commented line seemed to visibly influence the embeddings generated by BERT4Code Only for one of the modified lines, its original counterpart was returned as the top recommendation. However, since the line was a long (113 characters), inline comment, by removing the "//" we made it an invalid C/C++ line ("// Adapted from sample code in https://raw...") and difficult to match by any other line in the dataset. For the remaining lines, when a non-commented line was turned into a commented line, the suggestions provided by ACoRA became mostly commented lines. For instance, for the line "return 11_" modified to "// return 11_", the top recommendation was "// Hex dump -x" (both lines have four proceeding spaces and one space between "//" and the following token). The original line was ranked as the 30,903rd suggestion. Similarly, when we converted a commented line into a non-commented line, the suggested lines were also non-commented lines. For instance, for the line "/* packet_list.cpp" changed to "packet_list.cpp", the top suggestion was "cfutils.h" (the following suggestions were also mainly names of files). Interestingly, only at position 22 of the ranking was the first valid C/C++ code line consisting of an include statement ("#include "packet-ssl-utils.h"") that also contained the name of a file (with the word "packet"). The original line was returned as the 5,464th recommendation.

code_api — only the variant of ACoRA using CodeBERT did not top-ranked all the modified lines. It struggled with finding modified lines for " col_append_fstr(pinfo->cinfo, COL_INFO, " [zstd decompression failed]")_" and "decompress_lz4(tvbuff_t *tvb, packet_info *pinfo _U_, int offset, int length, tvbuff_t **decompressed_tvb, int *decompressed_offset)". In the case of the second line the modified line appeared at the 75th position of the ranking.

code_naming — ACoRA using CodeT5+ suggested nine out of ten modified lines in the first place while the remaining one was the 2nd recommendation. For other CodeEmbedders, in most cases, the original lines were either provided as a top suggestion or within the top 2 or 4 lines. All the perfect matches were lines that did not include method-/function call, e.g., "#define USBLL_POISON 0xFE ." For method calls, we observed that changing the casing was against the convention used by the Wireshark community. The suggested lines had similar structures and often included calls to function with similar identifiers. For instance, for the line "prefs_register_enum_preference(smpp_module, "decode_sms_over_smpp"," that we modified to Prefs_Register_Enum_Preference(smpp_module, "decode_sms_over_smpp",", the top suggestion was "prefs_register_uat_preference(someip_module, "_udf_someip_parameter_list", "SOME/IP Parameter List",".

PREVIEW

`code_design` — `ACoRA` using `CodeT5+` or `GraphCodeBERT` correctly suggested all ten modified lines as the most similar to the original ones. The variant using `CodeBERT` missed one of the modified lines and ranked it at 173rd place. For `BERT4Code`, six out of ten lines a change made to an identifier or adding/removing a keyword (adding `else` to an `if` statement; removing a `static` keyword) resulted in suggesting the original line at the top of the ranking. The lowest ranking position (219) was observed for an `if` statement "`if ( skipped > 0 )`" converted to a `while` statement "`while ( skipped > 0 )`" (the same as the one missed by `CodeBERT`). However, the top suggestion seemed more adequate than the original line since it was also a `while` statement "`while (cert_list_length > 0)`." Therefore, once again, it is not clear whether, in this case, the most similar line is the modified line or the one suggested by `ACoRA`.

`code_compatibility` — `ACoRA` variants using `BERT4Code` and `CodeT5+` were able to correctly indicate the modified lines. However, the two remaining `CodeEmbedders` missed only one line each that were ranked at positions 101 and 373 by `CodeBERT` and `GraphCodeBERT`, respectively.

5.1.2. RQ2: Relevance of recommendations – Wireshark

We based the evaluation for Wireshark on the dataset containing lines from 3,475 revisions. We wanted to balance the number of entries in *reference* and *validation* databases. Therefore, we added the first 1,760 revisions (40,430 comments) to the former database and the remaining 1,715 revisions (46,850 comments) in the latter one. We made the split timewise (2 years each).

We performed three analyses using different thresholds for matching lines belonging to the *reference* and *validation* databases. The thresholds were calculated based on the distance measured between the `CodeEmbedder`'s embedding vectors representing the lines within the *reference database* (the distance to the closest line in the *reference database* other than the line itself). Such a strategy allowed us to avoid biasing the observations by choosing such thresholds arbitrarily. Finally, we applied the filtering criteria described in Section 4. As a result, we obtained 1,582 recommendations to be analyzed.

Figure 6 shows the quality of `ACoRA`'s recommendations (measured using the O measure) for `ACoRA` using `CodeT5+` (plots for the remaining `CodeEmbedders` are available in the reproduction package) depending on the maximum distance threshold between the recommended and original lines, while the mean O values for all `CodeEmbedders` are presented in Table 3. When we set the threshold to the 10th percentile of the distances in the *reference database*, 99.6% of suggestions were relevant (only true-positive suggestions), however, the threshold limited the number of recommendations to 232 lines only (ca. 15% of all recommendations). As we increased the distance threshold to the 50th and 100th percentile, the number of relevant recommendations decreased to 82.8% and 40.7% while the number of irrelevant recommendations increased to 5.4% and 15.9%, respectively. This shows, unsurprisingly, that the possibility of providing correct recommendations for reviewers strongly depends on the contents of the *reference database*. However, even for the worst-case scenario (i.e., always recommending the comment of best-matching line in the *reference database*), `ACoRA` was able to provide at least partially relevant recommendations for 84.1% of the cases. These observations were consistent among other `CodeEmbedders`, with only minor differences in their accuracy of recommendations or the number of recommended lines depending on the threshold. We observed a trade-off between the number of recommended lines and the

PREVIEW

number of correct suggestions. Therefore, we cannot firmly state that either of the models is unanimously superior.

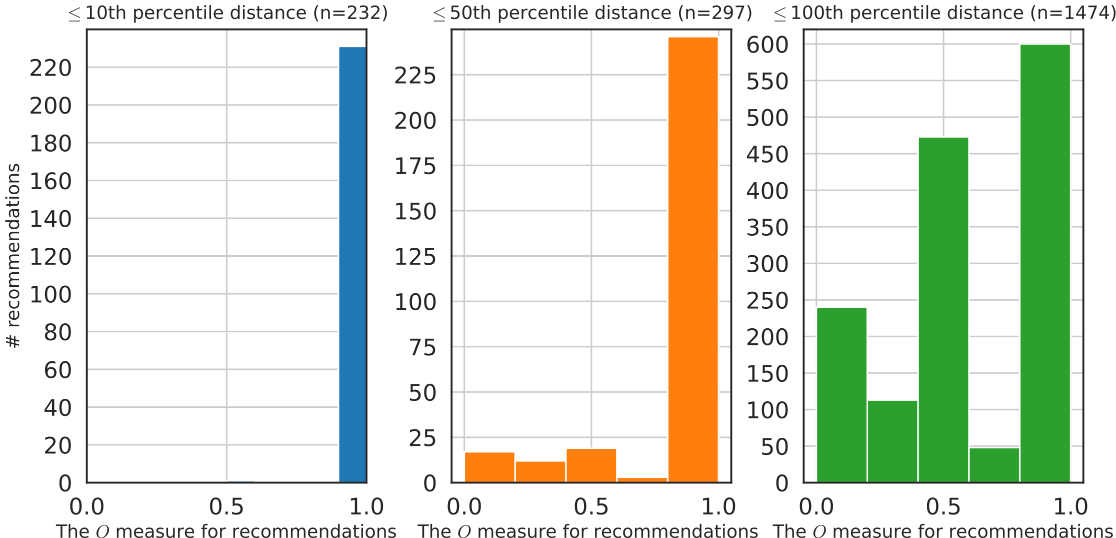

Figure 6. Evaluation of the overlap of recommended vs. actual categories for Wireshark (`CodeT5+`).

Table 3. Evaluation of the overlap of recommended vs. actual categories for Wireshark (all `CodeEmbedders`)

CodeEmbedder	≤ 10th perc. dist.		≤ 50th perc. dist.		≤ 100th perc. dist.	
	n	mean O	n	mean O	n	mean O
BERT4Code	233	0.98	465	0.78	1582	0.61
CodeBERT	233	0.99	366	0.83	1553	0.60
GraphCodeBERT	237	0.99	324	0.85	1609	0.61
CodeT5+	232	0.998	297	0.88	1474	0.61

5.2. Cloudera

5.2.1. RQ1: Finding similar lines – Cloudera

We modified up to 10 lines for each of the categories in our taxonomy and searched for similar lines in the dataset of 35,000 lines of code coming from Cloudera. Figure 7 presents the distributions of ranking positions of the original lines recommended by `ACoRA` using `CodeT5+` while Table 4 summarizes the results for all `CodeEmbedders`. Similar to Wireshark, most of the original lines were found as the most similar to their modified counterparts.

`config_commit_patch_review` — most of the lines belonging to this category were comments. Even for the recommendations having the original line ranked at 11+ position, the top suggestions seemed justifiable. For instance, the top recommendation for the line "`// initial transaction.`" modified by adding an e-mail address at the end was "`/// @note The replication factor should be an odd number and range in`"—the presence of `@` that was in the added e-mail address could have made the line more similar than the

PREVIEW

Table 4. Similarity of modified lines to their original counterparts for Cloudera (all `CodeEmbedders`).

`CodeEmbedder` / taxonomy category	**original line similarity rank**										
config_commit_patch_review	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	7	1	1	-	-	-	-	-	-	-	2
`CodeBERT`	9	-	-	-	-	-	-	-	-	-	2
`GraphCodeBERT`	8	-	-	-	-	-	-	-	-	-	3
`CodeT5+`	8	1	1	-	-	-	-	-	-	-	1
code_logic	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	7	-	-	-	-	-	1	1	-	-	-
`CodeBERT`	9	-	-	-	-	-	-	-	-	-	-
`GraphCodeBERT`	9	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	9	-	-	-	-	-	-	-	-	-	-
code_data	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	8	-	-	-	-	-	-	-	-	-	2
`CodeBERT`	9	-	-	-	-	-	-	-	-	-	1
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-
code_style	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	4	2	1	-	-	-	-	-	-	-	2
`CodeBERT`	9	-	-	-	-	-	-	-	-	-	-
`GraphCodeBERT`	8	-	-	-	-	-	-	-	-	-	1
`CodeT5+`	9	-	-	-	-	-	-	-	-	-	-
code_doc	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	2	-	-	-	-	-	-	-	-	-	7
`CodeBERT`	5	-	-	-	-	-	1	-	-	1	2
`GraphCodeBERT`	8	-	1	-	-	-	-	-	-	-	-
`CodeT5+`	9	-	-	-	-	-	-	-	-	-	-
code_io	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	6	1	-	1	-	-	-	-	-	-	2
`CodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	9	-	-	-	1	-	-	-	-	-	-
code_api	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	10	-	-	-	-	-	-	-	-	-	-
`CodeBERT`	6	-	1	-	-	-	-	-	-	-	3
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-
code_naming	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	5	-	1	-	-	-	-	-	-	1	2
`CodeBERT`	6	-	2	-	-	-	-	-	-	-	1
`GraphCodeBERT`	7	-	2	-	-	-	-	-	-	-	-
`CodeT5+`	9	-	-	-	-	-	-	-	-	-	-
code_design	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	8	-	-	-	-	-	-	-	-	-	1
`CodeBERT`	8	1	-	-	-	-	-	-	-	-	-
`GraphCodeBERT`	9	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	9	-	-	-	-	-	-	-	-	-	-
compatibility	1	2	3	4	5	6	7	8	9	10	11+
`BERT4Code`	6	1	-	2	-	1	-	-	-	-	-
`CodeBERT`	6	2	-	-	1	-	-	-	-	-	1
`GraphCodeBERT`	10	-	-	-	-	-	-	-	-	-	-
`CodeT5+`	10	-	-	-	-	-	-	-	-	-	-

PREVIEW

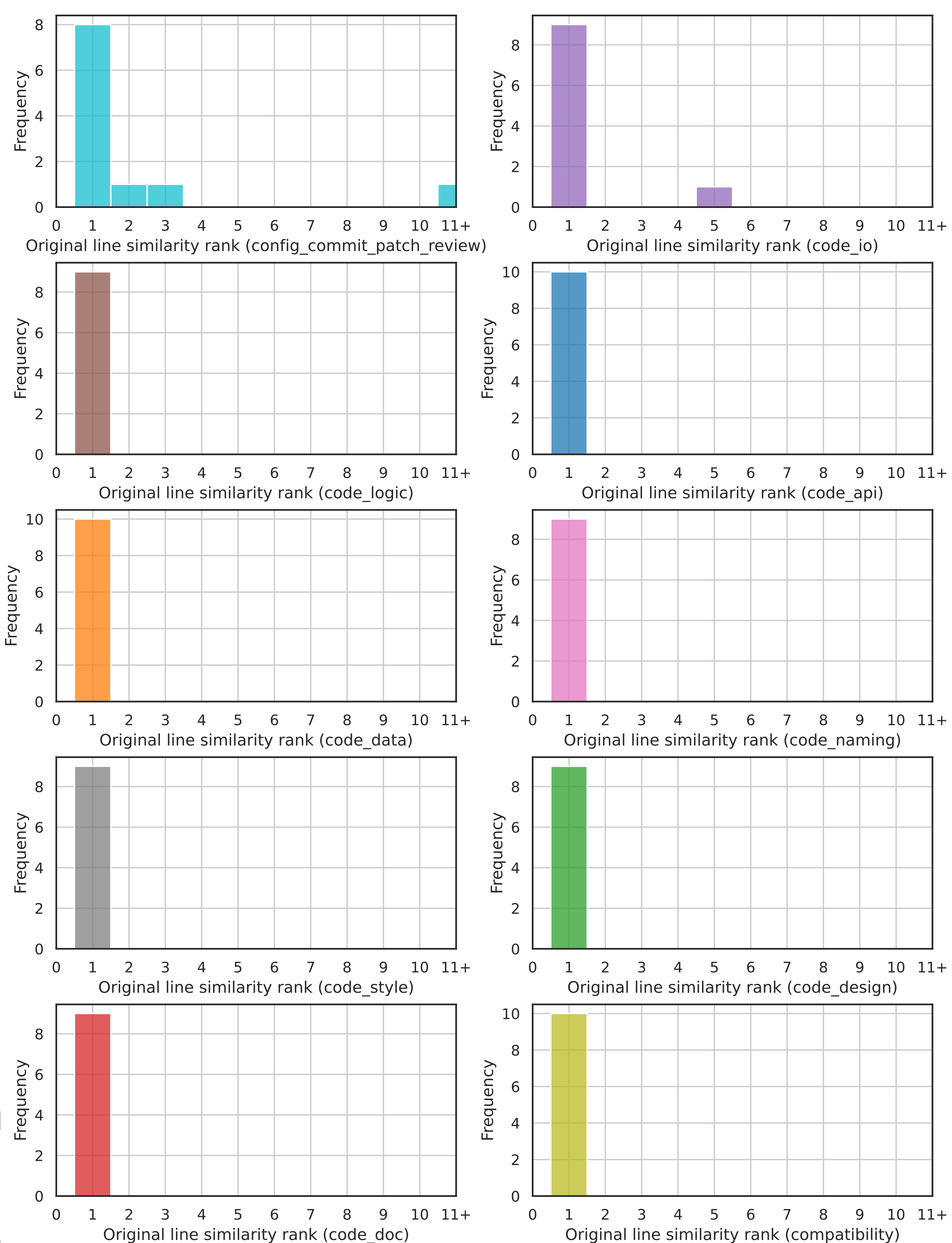

Figure 7. Similarity of modified lines to their original counterparts for Cloudera (`CodeT5+`).

original line. The second case was the line "`ASSERT_FALSE(empty .has_user());`" modified by negating the parameter of the call (`!empty...`). Although the original line was ranked at the 16th position, the other 13 top suggestions were also the assert functions calls.

code_logic — ACoRA variants using CodeT5+, CodeBERT, and GraphCodeBERT correctly indicated all modified lines as top suggestions, while ACoRA using BERT4Code missed two lines. The first one was the line "if (schema_elem.__isset.field_id) {" modified by negating the condition (!). The first six recommendations were also if statements with negations, e.g., the top recommendation was the line"if (!step.has_add_column()) {". The second line was "return percent/4;" with the operator changed to multiplication (*). All recommendations were return statements. However, the top recommendation contained a pointer reference instead of the multiplication "return *this;"

code_data — ACoRA variants using CodeT5+ and GraphCodeBERT correctly indicated modified lines, however, the two remaining CodeEmbedders missed only up to two lines. For BERT4Code two missed cases were the same line "int percent = 0;" modified by changing the declared type to double and string.

code_style — ACoRA variants using CodeT5+ and CodeBERT correctly suggested all modified lines, while GraphCodeBERT missed one of the lines and recommended it as the 39th similar line. For ACoRA using BERT4Code, we observed more original lines recommended at high-ranking positions than for Wireshark, however, also the nature of changes made to the lines was slightly different. Two lines for which the top recommendations were the original lines were modified by removing a single indenting space (from six to five). Another one was adding an additional space between the and operator (&&) and function call ("&&␣std::find..." to "&&␣␣std::find..."). Our general observation was that the the bigger the difference in the number of indention spaces the less similar the original and modified lines were. The extreme case was the line "␣␣␣␣␣␣␣␣int num_micro_batches = " with all indention spaces removed (the original line was provided as the 282nd recommendation).

code_doc — ACoRA variants using CodeT5+ and GraphCodeBERT correctly indicated ten and nine out of ten modified lines and performed visibly better than the remaining variants of ACoRA using CodeBERT and BERT4Code. For these two models, we made a similar observation to Wireshark that converting between commented and non-commented lines made them perceived as non-similar. For BERT4Code, two exceptions were (1) an inline comment changed to a single-line block comment and (2) an inline comment with preceding // modified to /// for which the original lines were provided as the first suggestions.

code_io — ACoRA variants using CodeBERT and GraphCodeBERT correctly indicated all ten modified lines as the most similar to the original ones. ACoRA using CodeT5+ made only one mistake, however, the top-suggested line was a similar logging statement to the original one. For ACoRA using BERT4Code changing the names to synonyms led to most modified lines being recognized as the most similar to their original counterparts. The two extreme cases were the lines "LOG(ERROR) << s;" and "return server_->Init();" modified by changing ERROR to PROBLEM and server to computer. Interestingly, for the latter, one of the top suggestions was the line "return client->DeleteTable(p.table_name);" containing a word client used in a similar context, which could mean that the word computer was perceived by the model to be more similar to the word client than to the word server.

code_api — three ACoRA variants using BERT4Code, CodeT5+, and GraphCodeBERT correctly indicated all ten modified lines as the most similar to the original ones. The exception was the ACoRA variant using CodeBERT which correctly suggested six lines but in three cases the modified line appeared beyond the first ten positions of the ranking (592, 328, and 943).

PREVIEW

`code_naming` — only `ACoRA` variant using `CodeT5+` was able to correctly indicate all modified lines as the most similar to the original ones. The variants using `GraphCodeBERT` and `CodeBERT` were slightly worse. For the variant of `ACoRA` using `BERT4Code`, we made similar observations as for Wireshark. However, we observed three lines for which the original lines were recommended as a 10+ option. The first one was changing the casing in the name of `struct` "`struct TestData {`" (`TestData` to `test_data`). The second one was changing "`LOG.debug(...`" to "`log.debug(...`". Interestingly, it turned out that a line "`log.info(...`" was found as one of the top three recommendations. The third line contained a call to a function "`QUERY_OPT_FN(...`" which identifier was changed to "`query_opt_fn(...`." These changes had the biggest impact on similarity among the three lines and the original line was ranked at the 307th position.

`code_design` — `ACoRA` variants using `CodeT5+` and `GraphCodeBERT` correctly provided all the modified lines as top-suggestions. Both `BERT4Code` and `CodeBERT` did not provide correct suggestions for one line—"`FragmentState* fragment_state;`" modified to "`RuntimeState* fragment_state;`". Interestingly, the top suggestions were nearly identical lines "`RuntimeState* state;`" and " `Server`
`State* server;`.

`compatibility` — again, `ACoRA` variants using `CodeT5+` and `GraphCodeBERT` correctly provided all the modified lines as top-suggestions. For `BERT4Code`, we made a similar observation as for Wireshark that adding a suffix `_v2` to identifiers allowed for recognizing the original lines as very similar. Even for the cases where the top suggestion was not the original line, all of the recommended lines had a very similar structure. For instance, for the modified line "`import java.io.IOException_v2;`" all six suggestions were imports with "`import java.io.InputStreamReader;`" as the top recommendation.

5.2.2. RQ2: Relevance of recommendations – Cloudera

We followed the procedure described in Section 4 to select a sample of 29,599 lines as a *reference database* and 51,024 lines as the *validation database*. For Cloudera, we focused on scaling up the size of the validation dataset. The *reference database* was extracted from the 74K review comments that we initially fetched from the Gerrit instance, while the lines included in the *validation database* were extracted from the remaining (ca. 174K) comments fetched in later runs. Next, we applied the filtering criteria, which resulted in 4,738 recommendations to be analyzed.

Table 5 presents the mean O values for all `CodeEmbedder`s depending on the similarity distance thresholds. The mean O value for the 10th percentile ranges between 0.69 and 0.89, dropping to 0.53–0.57 for the 100th percentile. The plot in Figure 8 shows that when all the recommendations were considered, the `ACoRA` variant using `CodeT5+` provided irrelevant suggestions in 23.7% of the lines (1,158 lines), i.e., the overlap between the actual categories and the recommended categories was 0%. For 76.3% of the lines (3,721 lines), `ACoRA` recommended at least one of the categories correctly. For 35.0% of the lines (1,707 lines), the match was fully correct, i.e., the recommended categories were the same as the actual categories of the comment. Once we lowered the distance threshold (a minimum distance between the embedding vectors), we observed that the relevance of the recommendations increased. For the threshold equal to the 50th percentile of the distances in the *reference database*, the percentage of relevant recommendations increased to 46.5%, and for the most strict threshold corresponding to the 10th percentile, 72.4% of suggestions were relevant.

PREVIEW

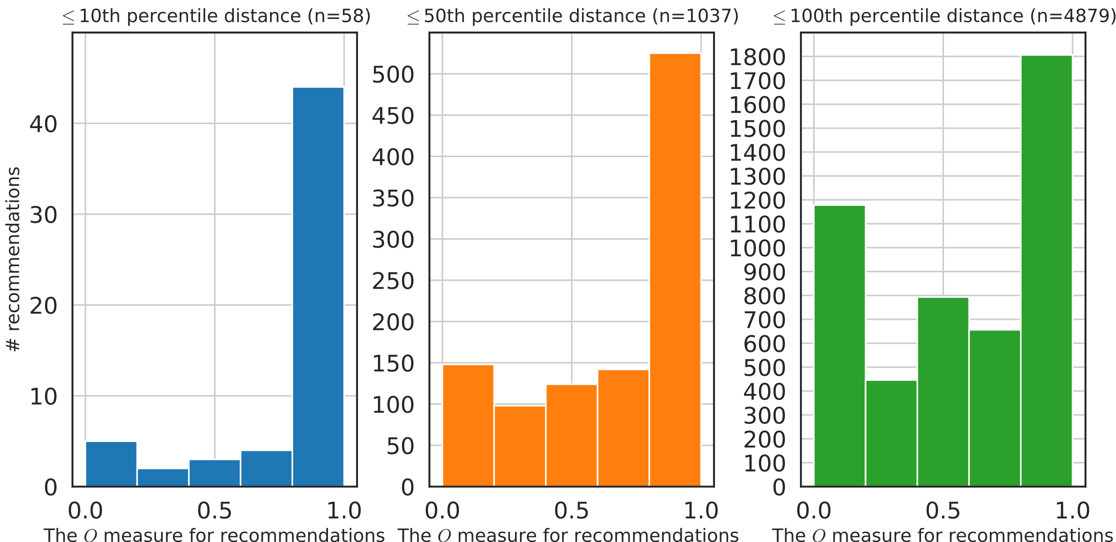

Figure 8. Evaluation of the overlap of recommended vs. actual categories for lines for Cloudera (`CodeT5+`).

Again, we observed a trade-off between the accuracy of the `ACoRA` and the number of recommended lines depending on the choice of a `CodeEmbedder` model.

Table 5. Evaluation of the overlap of recommended vs. actual categories for Cloudera (all `CodeEmbedders`)

`CodeEmbedder`	**≤ 10th perc. dist.**		**≤ 50th perc. dist.**		**≤ 100th perc. dist.**	
	n	mean O	n	mean O	n	mean O
`BERT4Code`	139	0.89	1997	0.57	4738	0.53
`CodeBERT`	85	0.69	1816	0.61	4343	0.54
`GraphCodeBERT`	62	0.73	1599	0.59	4513	0.55
`CodeT5+`	58	0.83	1037	0.68	4879	0.57

We can conclude that the recommendations provided by `ACoRA` are relevant for the majority of cases (76.3%) and that the differences are often observed in a few categories (1–3), even when no similarity threshold is used.

5.3. Evaluation at the industrial partner

When evaluating the tool at the industrial partner, we analyzed a number of commits:

1. Comment on keyword `const` in a function parameter list. The reviewer asked for removing the keyword, i.e., changing the parameter to a non-constant variable. `ACoRA` identified this line as well, with the recommendation that `code_logic` should be investigated. In the database of examples, there was only one similar comment, but it was used in another context; therefore most of the examples were not relevant.
2. Reviewer questioned the re-location of a code fragment – he/she asked whether the code was moved correctly. `ACoRA` could not isolate the code fragment which was relevant, instead identified most lines in the commit as problematic, with different suggestions. The examples were mostly not relevant.

PREVIEW

3. A reviewer commented on the name of a function, asking for a change of the name (not the entire signature). `ACoRA` identified the line with the name of the function as problematic, but the suggestion was to fix the **code_logic** instead of `code_naming`.
4. Reviewer asked to change a number of `#define` statements to `const`. `ACoRA` identified all such statements correctly, e.g. `#define x 1` and omit statements which should be omitted, e.g. `#define (x | y)`, which cannot be changed to a `const`. There were no relevant examples of the comments database and therefore the provided examples were not relevant.

During the discussion, we identified two major challenges for making this type of tool usable at the company.

The first challenge was the ability to capture the context of the code—not the lines before or after the commented line, but the ability to trace what has been done to the line in its context. For example:

– whether the line was newly added as part of the entire block or just a single line,
– whether the line was added as part of a large commit (e.g. more than three files were changed),
– whether the code block where the line is located has been in the code-base from the beginning or was added in a recent few commits (if it was not added in the commit-under-review).
– what was previously discussed in this code block, e.g. whether there was a discussion about a design solution for this block, naming conventions, etc.

Understanding the context of a reviewed line in this way would mimic the understanding of the context of the code reviews by human reviewers.

The second challenge was the ability to use meta-data in model training and prediction. The meta-data could contain the information of the context as in the first challenge, and the meta-information about the committed code fragment – its complexity, size, type of changed code (e.g., the role of the class/module), type of the system (e.g., safety-critical vs. web back-end). This information is available to the code reviewers as they know the system, but it is not available to tools like `ACoRA` (or even systems like CodeX [53]).

6. Implications for practitioners

The key takeaway from our study for practitioners is that a complex problem of automatic code review and repair can be simplified to a simpler problem of searching for similar lines to those under review and providing a summary of issues previously raised by reviewers to increase the accuracy of automatic code review support under the cost of increasing human involvement in the review process. Another general lesson from our study is that setting a similarity threshold while searching for similar lines of code based on language model embeddings is a critical success factor. Our study shows that such a threshold should not be set arbitrarily but should come from understanding how similar or different the code is in the considered codebase. To tackle this problem, we propose determining the threshold by sampling a codebase and using a percentile-based approach that allows for balancing recall (higher percentile) and precision (lower percentile). Finally, we show that the concept proposed by `ACoRA` can be implementedre using different `CodeEmbedder` models. Although neither of the studied Transformer-based models turned out to be unanimously superior, we

PREVIEW

suggest using large pre-trained models (e.g., CodeT5+) as a default option, however, in the scenario where a codebase is unique (e.g., contains only in-house developed components or developers use unique, company-specific coding guidelines), one might consider pre-training a BERT4Code model from scratch, as it can be easily done even using standard graphics processing units (GPU).

7. Validity evaluation

In our validity analysis, we use the framework advocated by Wohlin et al. [54], complemented with the threats specific to studies embedded in external context as prescribed by Weringa [17].

In the category of *construct validity*, our major threat is the use of machine-learning language models to extract features. Although it is modern technology, using word embedding networks does not allow us to construct a vocabulary manually. This means that we do not know whether language constructs important for programmers (e.g., keywords) are important for the neural network too. Our mitigation strategy is to study different techniques for feature extraction (presented in [55] and [56]). We have also examined the results of the similarity of lines, manually in this paper, in order to understand whether this is a threat in our case.

Another construct validity threat is our classification of comments. Although it is based on our previous studies [16] and the systematic review by [38], it is a generalization of a natural language in a specific context. To minimize the risk of bias towards a project-specific language, we used pre-trained models that provide the same classifications based on several projects.

The most important threat to *conclusion validity* is measuring the relevance of the suggestions provided by ACoRA. A single comment can relate to several issues belonging to different taxonomy categories. This means that there is a risk that this is a multi-label classification problem. Unfortunately, the multi-label classification makes the evaluation of ACoRA suggestions challenging, as they must be assessed across multiple categories simultaneously. Simplifying this assessment to a binary evaluation for individual categories might seem straightforward, but it fails to capture the nuanced reality of multi-faceted feedback. Therefore, we used a measure that captures the overlap between categories to tackle this issue. Also, to minimize the threat of making wrong conclusions, we manually analyzed a sample of suggestions with non-overlapping categories between suggested and actual comments.

When conducting the study, we chose to evaluate it at a company. We've selected one of our collaboration partners based on the domain—embedded systems, long experience with programming, and access to experienced architects (>10 years). However, there could be a threat to *internal validity* associated with how we worked with the company. Since ACoRA uses source code from a company to operate, we set it up to connect directly to the company's code repository at their premises. We extracted code changes and the associated comments, as prescribed by the process of using ACoRA. The time for that was limited due to access to the premises and experts, and therefore, we could not conduct this evaluation for an extensive period of time. We collected and analyzed the data on-premises while presenting the results off-site (via MS Teams). This could mean that there is a risk that we missed an important aspect of the evaluation or that we did not manage to select the most optimal code fragments to discuss (the selection was random).

PREVIEW

Finally, since our evaluation is performed on two open source projects and one industrial project, there is a risk of being too specific, i.e., an *external validity* risk. We have considered this and therefore asked the company about this specific aspect, as well as we manually examined the results (random samples of results). We concluded that the company or project is not the decisive factor but the availability and quality of the data. We applied ACoRA on two other projects (one industrial and one open source), where we extracted a handful of comments only. The low number of data points did not provide any results, and therefore, they are not included, but we've learned about the limitations and, therefore, claim that the results are generalizable. They are generalizable to contexts where the number of commented code fragments is >1,000 and when the comments are linked (in the tool) to code fragments and not to entire commits/patches. Also, we assume that the comments and lines of code in the historical database are similar to the lines in the code under review—thus, we generalize our findings to an intra-organization/process application of the proposed approach. More studies are needed to find if we can generalize findings across different projects. Such studies are planned for our future work. Also, we suspect that the accuracy and usefulness of the proposed approach might decrease even in an intra-project application scenario if the context changes visibly over time, making the historical database of comments invalid (e.g., the nature of the project changes significantly, and quality standards evolve). Another threat to external validity concerns the number of code lines selected per comment-taxonomy category while investigating the tool's ability to find similar lines for each category. We randomly queried the dataset to obtain representative samples of lines of code and comments belonging to particular taxonomy categories (within our dataset); however, we are not able to assess how well they cover the whole spectrum of lines/comments in these categories. Finally, we identified one more threat to external validity that regards the selection of "mutation" operations we applied to modify lines of code. We made these choices subjectively to ensure that they regard those code constructs that are decisive while categorizing a given line of code into a particular taxonomy category. However, we are aware that the variety of such code constructs that seem valid in the context of each category goes beyond the examples we provide.

8. Conclusions and future work

Code reviews are an integral part of modern software development, usually being integrated with the continuous integration/deployment pipelines. Although there are tools that automatically check the quality of source code, code reviews are still needed—they provide the ability to comment on design aspects that cannot be formalized into checking rules, they provide the ability to discuss design choices, and, not least important, they are a way of onboarding new project members.

Although it has already been found that we can pinpoint which code fragments (even down to a single line of code) would attract attention from reviewers, there is little support for providing suggestions on what to focus on. In this work, we address this problem by designing and evaluating a tool for automatically providing these suggestions based on the previous review discussions available in code repositories. The tool uses a machine-learning based language model and searches for similar lines of code to those under review that were previously commented on. It analyzes the previous reviewers' comments to indicate the aspects of the code the reviewer should focus on while reviewing a given fragment of code. The suggestions are based on company/community-specific culture and provide a

PREVIEW

way to speed up the review process while allowing new team members to understand the code and participate in the code review discussions.

By using two open-source projects and one industry project, we studied to which degree it is possible to provide code reviewers with a suggestion on what they should focus on when reviewing a given code fragment. The results show that the tool can give fully correct suggestions (only true positives) in 35%–41% of the fragments and partially correct suggestions in 76.3%–84.1% of the fragments. Compared to the state-of-the-art tools for code repair, this is higher but requires human intervention (it is the reviewer who has to review the code in the end). Also, we showed that one can control the recall and precision of such recommendations by changing a similarity threshold between the code fragments (for the distance between the line-embedding vectors). By setting the threshold to the 10th percentile of the distances in the reference dataset, we were able to increase the correctness of recommendations even to 72%–99%, however, at the cost of sacrificing the number of recommendations provided by `ACoRA`. Therefore, a key takeaway from our study for practitioners is that a complex problem of automatic code review and repair can be simplified to the problem of finding previously commented-on lines of code that are similar to those under review and summarizing the issues raised by reviewers. This approach allows for increasing the accuracy of automatic code review support, however, at the cost of increasing human involvement in the review process.

In future work, we plan to conduct a deeper analysis and comparison of machine-learning language models and code representations to study their impact on the accuracy of `ACoRA`. We plan to expand this study to design a pre-configured tool set-up to identify specific aspects, e.g., security vulnerabilities in source code and SQL injection checks, and evaluate them in an industrial context. In particular, we want to create portable (between contexts) reference databases designed to detect certain types of issues in the code.

Acknowledgements

This research has been supported by Software Center (www.software-center.se), Chalmers | University of Gothenburg and by National Science Centre, Poland within the research project "Source code representations for machine-learning-based identification of defective code fragments" (OPUS 21), registered with the no. 2021/41/B/ST6/02510.

The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] P.C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.
[2] L. MacLeod, M. Greiler, M.A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, Vol. 35, No. 4, 2017, pp. 34–42.
[3] M.E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 182–211.
[4] L. Milanesio, *Learning Gerrit Code Review*. Packt Publishing Ltd, 2013.
[5] M. Meyer, "Continuous integration and its tools," *IEEE Software*, Vol. 31, No. 3, 2014, pp. 14–16.

PREVIEW

[6] M. Staron, W. Meding, O. Söder, and M. Bäck, "Measurement and impact factors of speed of reviews and integration in continuous software engineering," *Foundations of Computing and Decision Sciences*, Vol. 43, No. 4, 2018, pp. 281–303.
[7] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs. How the current code review best practice slows us down," in *IEEE/ACM 37th International Conference on Software Engineering*, Vol. 2. IEEE, 2015, pp. 27–28.
[8] F. Huq, M. Hasan, M.M.A. Haque, S. Mahbub, A. Iqbal et al., "Review4Repair: Code review aided automatic program repairing," *Information and Software Technology*, Vol. 143, 2022, p. 106765.
[9] M. Hasan, A. Iqbal, M.R.U. Islam, A. Rahman, and A. Bosu, "Using a balanced scorecard to identify opportunities to improve code review effectiveness: An industrial experience report," *Empirical Software Engineering*, Vol. 26, No. 6, 2021, pp. 1–34.
[10] M. Staron, M. Ochodek, W. Meding, and O. Söder, "Using machine learning to identify code fragments for manual review," in *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 513–516.
[11] D.S. Mendonça and M. Kalinowski, "An empirical investigation on the challenges of creating custom static analysis rules for defect localization," *Software Quality Journal*, 2022, pp. 1–28.
[12] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.
[13] N. Fatima, S. Nazir, and S. Chuprat, "Knowledge sharing, a key sustainable practice is on risk: An insight from modern code review," in *IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*. IEEE, 2019, pp. 1–6.
[14] A. Hindle, E.T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, Vol. 59, No. 5, 2016, pp. 122–131.
[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones et al., "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.
[16] M. Ochodek, M. Staron, W. Meding, and O. Söder, "Automated code review comment classification to improve modern code reviews," in *International Conference on Software Quality*. Springer, 2022, pp. 23–40.
[17] R. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*, 2014. [Online]. http://portal.acm.org/citation.cfm?doid=1810295.1810446
[18] M. Allamanis, E.T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, Vol. 51, No. 4, 2018, pp. 1–37.
[19] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, Vol. 3, No. POPL, 2019, pp. 1–29.
[20] J. Devlin, M.W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[21] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng et al., "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
[22] B. Roziere, M.A. Lachaux, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," *arXiv preprint arXiv:2102.07492*, 2021.
[23] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser et al., "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.
[24] F. Huq, M. Hasan, M.M.A. Haque, S. Mahbub, A. Iqbal et al., "Review4repair: Code review aided automatic program repairing," *Information and Software Technology*, Vol. 143, 2022, p. 106765. [Online]. https://www.sciencedirect.com/science/article/pii/S0950584921002111
[25] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk et al., "Using pre-trained models to boost code review automation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2291–2302.
[26] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.
[27] A. Ram, A.A. Sawant, M. Castelluccio, and A. Bacchelli, "What makes a code change easier to review: An empirical investigation on code change reviewability," in *Proceedings of the 26th*

PREVIEW

ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 201–212.

[28] Y. Arafat, S. Sumbul, and H. Shamma, "Categorizing code review comments using machine learning," in *Proceedings of Sixth International Congress on Information and Communication Technology*. Springer, 2022, pp. 195–206.

[29] Z. Li, Y. Yu, G. Yin, T. Wang, Q. Fan et al., "Automatic classification of review comments in pull-based development model," in *International Conferences on Software Engineering and Knowledge Engineering*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2017.

[30] Z.X. Li, Y. Yu, G. Yin, T. Wang, and H.M. Wang, "What are they talking about? Analyzing code reviews in pull-based development model," *Journal of Computer Science and Technology*, Vol. 32, 2017, pp. 1060–1075.

[31] L. Yang, J. Xu, Y. Zhang, H. Zhang, and A. Bacchelli, "EvaCRC: evaluating code review comments," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 275–287.

[32] J.K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[33] R. Brito and M.T. Valente, "RAID: Tool support for refactoring-aware code reviews," in *IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 265–275.

[34] R. Tufano, O. Dabić, A. Mastropaolo, M. Ciniselli, and G. Bavota, "Code review automation: Strengths and weaknesses of the state of the art," *IEEE Transactions on Software Engineering*, 2024.

[35] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Commentfinder: a simpler, faster, more accurate code review comments recommendation," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 507–519.

[36] D. Badampudi, R. Britto, and M. Unterkalmsteiner, "Modern code reviews – Preliminary results of a systematic mapping study," *Proceedings of the Evaluation and Assessment on Software Engineering*, 2019, pp. 340–345.

[37] D. Badampudi, M. Unterkalmsteiner, and R. Britto, "Modern code reviews – Survey of literature and practice," *ACM Transactions on Software Engineering and Methodology*, Vol. 32, No. 4, 2023, pp. 1–61.

[38] N. Davila and I. Nunes, "A systematic literature review and taxonomy of modern code review," *Journal of Systems and Software*, Vol. 177, 2021, p. 110951.

[39] H.A. Çetin, E. Doğan, and E. Tüzün, "A review of code reviewer recommendation studies: Challenges and future directions," *Science of Computer Programming*, Vol. 208, 2021, p. 102652.

[40] B. Roziere, M.A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, Vol. 33, 2020, pp. 20 601–20 611.

[41] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[42] M. Ochodek, M. Staron, D. Bargowski, W. Meding, and R. Hebig, "Using machine learning to design a flexible loc counter," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2017, pp. 14–20.

[43] M. Ochodek, R. Hebig, W. Meding, G. Frost, and M. Staron, "Recognizing lines of code violating company-specific coding guidelines using machine learning," *Empirical Software Engineering*, Vol. 25, No. 1, 2020, pp. 220–265.

[44] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi et al., "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[45] I. Turc, M.W. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: On the importance of pre-training compact models," *arXiv preprint arXiv:1908.08962*, 2019.

PREVIEW

[46] H. Akoglu, "User's guide to correlation coefficients," *Turkish journal of emergency medicine*, Vol. 18, No. 3, 2018, pp. 91–93.

[47] J.N. Mandrekar, "Receiver operating characteristic curve in diagnostic test assessment," *Journal of Thoracic Oncology*, Vol. 5, No. 9, 2010, pp. 1315–1316.

[48] M. Staron, *Action research in software engineering*. Springer, 2020.

[49] S.K. Pandey, M. Staron, J. Horkoff, M. Ochodek, N. Mucci et al., "TransDPR: design pattern recognition using programming language models," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2023, pp. 1–7.

[50] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang et al., "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[51] Y. Wang, H. Le, A.D. Gotmare, N.D. Bui, J. Li et al., "CodeT5+: Open code large language models for code understanding and generation," *arXiv preprint*, 2023.

[52] V. Antinyan, M. Staron, A. Sandberg, and J. Hansson, "Validating software measures using action research a method and industrial experiences," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, 2016, pp. 1–10.

[53] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d.O. Pinto et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[54] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell et al., *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[55] K.W. Al-Sabbagh, M. Staron, M. Ochodek, R. Hebig, and W. Meding, "Selective regression testing based on big data: Comparing feature extraction techniques," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2020, pp. 322–329.

[56] M. Staron, W. Meding, O. Söder, and M. Ochodek, "Improving quality of code review datasets – Token-based feature extraction method," in *International Conference on Software Quality*. Springer, 2021, pp. 81–93.

PREVIEW