

A Multi-Step Learning Approach to Assist Code Review

Oussama Ben Sghaier

University of Montreal

Montreal, Canada

oussama.ben.sghaier@umontreal.ca

Houari Sahraoui

University of Montreal

Montreal, Canada

houari.sahraoui@umontreal.ca

Abstract—Modern code review is a process for early detection and reduction of issues, which assists in ensuring the quality of the source code, detecting anomalies, and identifying potential improvements. However, this is a highly manual activity that requires a lot of resources and time. Recent research has addressed these problems by attempting to entirely automate this task (i.e., generating code reviews). However, we do believe that dismissing the reviewer from this process is not the best option in terms of its optimal functioning, especially considering the high error rates in the proposed approaches. Furthermore, this full automation is still too far to achieve given the complexity of the task that requires human intelligence. In this work, we aim to assist the reviewer in the code review process. We propose an approach for detecting the type of issue and locating parts of the code that need to be revised by developers. In the first phase, we propose a meta-learner that combines a learning-based model and a knowledge-based model to predict the type of issue from the review comment. Then, we use this component to create and label a large dataset composed of quadruplets <original code, review, issue type, revised code>. We use this data set to fine-tune a pre-trained language model to predict the types of issues (e.g., naming, resource handling, etc.), that need to be addressed in the original code snippet. Furthermore, we fine-tune another pre-trained language model to locate these issues in the source code submitted by developers. We evaluate the performance of our approach using a test set not considered during the training. Our results show that our model accurately locates and predicts the types of issues.

Index Terms—Natural language processing, deep neural networks, modern code review, code analysis, recommender systems.

I. INTRODUCTION

Continuous integration (CI) is the practice of continuously inspecting, testing, and integrating source code written by developers who collaborate on a project [1]. Among the reported benefits of CI are better collaboration and code quality [2]. Code review is a common step in the continuous integration pipeline. It is a manual process where one or more developers inspect the code written by fellow developers [3], [4]. Developers regularly review each other's code to identify potential issues and improvements and maintain high code quality [5]–[7]. Reviewing includes writing comments pinpointing out to the specific change that introduced the issue. It might also recommend a way to fix it [8].

The predominantly manual nature of current source code development practices is a source of issue proneness (e.g., resource handling errors, documentation issues, etc.) [9]. Thus,

code review is one of the activities whose purpose is to identify these potential issues [10], [11]. A key benefit of regularly reviewing source code after each change is the potential early detection of erroneous and sub-optimal code related to that change.

Automating the different steps of the code review process remains a challenge. One area of research targets the early stages of this process by identifying potential issues using *static analysis* [12]. The proposed tools, called *linters*, define rules that represent security issues, stylistic errors, code quality, best practices, etc. and flag parts of the code that violate them. These rules are defined manually and require a lot of adaptation to cover most of the issues [13]. Moreover, these issues and best practices are variable over time and depend on many factors including software architecture, team, project, domain, etc. This stiffness of static analysis tools makes them less effective and less suited to widespread use in software projects [14]. A second research area assists the reviewers in the writing of review comments by suggesting examples of reviews that can be relevant to a given code snippet [15]. Although such suggestions can be useful, review comments are rarely generic and more often relate to a specific context. A third research area focuses on the later stages of code review by suggesting code changes from the code to review or from the review comments [16], [17]. In general, these approaches suggest changes without pointing a specific issue and without explanations.

Although these works are attempting to entirely automate code review, this goal still has not been achieved considering the complexity of this process and the high error rates in the proposed methods. This makes these tools less usable and less efficient. Since the activity of code review requires human intelligence to adapt to new issues and to suggest creative solutions, using intelligent assistants to supplement human work will probably result in greater advantages than its mere substitution [18], [19]. That is, we do believe that the best solution is not to entirely substitute the reviewer, but rather to assist him and help him doing his task easily and accurately.

In this paper, we focus on the issue *identification* (i.e., issue categorization and localization) to assist the reviewers during the code review process. This helps to avoid having to focus solely on the most common issues while neglecting some other important ones [8]. Additionally, as software projects tend to

grow incrementally and become more complex, the complexity of code review increases as the size of the code base grows [20], [21]. Consequently, the review process requires more time and more effort to identify and locate the issue in the source code [22].

To address the aforementioned challenges, we propose an approach that consists of the following contributions:

- We combine a learning-based model with a knowledge-based model to predict the type of issue based on the reviewer's comment. We use the produced model to label a large dataset that will serve as input for the second contribution.
- We fine-tune pre-trained state-of-the-art language models on a downstream task: predicting the type of issue from source code.
- We fine-tune pre-trained state-of-the-art language models on locating issues from an input code fragment.

This paper is organized as follows. Section II briefly explains the background. Section III details the different components of our proposed approach. Section IV reports the evaluation results and discusses some threats related to our experiments. Section V outlines some related work. Finally, Section VI concludes.

II. BACKGROUND

A. Code review

In a continuous integration setting, developers use version control systems (e.g., GitHub) to collaborate and manage software versions. That is, version control systems allow development teams to manage and keep track of changes made to the software codebase over time [1], [23]. Developers, working as a team, continuously write new source code and modify the existing source code. Figure 1 illustrates the common software development workflow in a continuous integration setting.

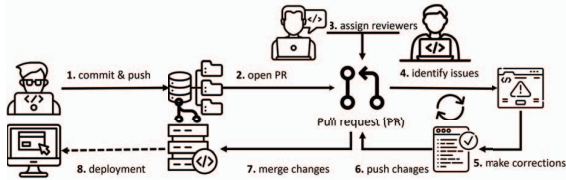


Fig. 1. Software development workflow in a continuous integration setting

A developer, who is working on implementing a new feature, makes local changes to the codebase. Then, he commits and pushes his changes to a shared repository ① to make the new changes available to collaborators. He can create a pull request ② to request that his changes be merged to the main branch (i.e., main version of the source code). Some reviewers are assigned to the pull request ③ to inspect the code changes made by the author before these changes are merged with the main branch. Reviewers identify and locate issues ④ that need to be solved by the code author (⑤ and ⑥). The revised version of the code, submitted by developers, should be reviewed again. Once these changes are approved by reviewers, they

are merged with the main code version, i.e., main branch ⑦, to be available to other developers (i.e., collaborators). If necessary, the project can be deployed ⑧, so that changes will be available to end-users.

B. Pre-trained language models

State-of-the-art language models, that are based on the *transformer architecture* [24], have shown outstanding performance in solving natural language problems. Transformers have several advantages over recurrent neural networks. They enable parallel computations which reduces training time and captures long-range dependencies efficiently. *BERT* [25], *GPT-3* [26], *XLNet* [27], *RoBERTa* [28] and *T5* [29] are examples of transformers that were pre-trained on tons of data to provide general and high-level representations of text. These pre-trained models could then be adapted (i.e., fine-tuned) on downstream tasks using specific datasets.

Transformers are sequence-to-sequence models based on the attention mechanism that takes into account the relationship between all the words in the sentence and not only the contextual words (e.g., previous and next words). The attention mechanism learns a weighting function indicating how much each element contributes to the prediction (i.e., importance of each input element in predicting the target).

C. Ensemble learning

Ensemble learning is a meta-approach to machine learning that combines multiple experts into a single model to have better predictive performance and to improve the robustness of the predictions [30], [31].

Stacking is an ensemble learning method that trains several types of experts on the same training set. Then, a meta-learner is used to learn how to best combine the predictions of the different experts. In this method, several experts (i.e., models) are trained on the same dataset. The predictions of these models are fed to a meta-learner (i.e., a machine learning model) to generate the final predictions [32]. Combining several machine learning models using ensemble learning techniques enables better performance and a more robust model.

III. PROPOSED APPROACH

In this section, we give an overview of our approach, present the datasets used in this work, and discuss each component of the approach.

A. Approach overview

Figure 2 shows an overview of the proposed approach that is composed of four main steps. The goal of the first step ① is to create and prepare the dataset. We start with an existing small labeled dataset of $\langle review, type \rangle$ couples in which the label *type* refers to the issue type pointed out by the review. Then, we have a large unlabeled dataset of $\langle original\ code, review \rangle$ pairs in which the labels (i.e., *type*) are missing. The *original code* variable refers to the source code submitted by developers for review. In ①, we aim to build and train a model on the small labeled dataset to label our large dataset with the types

of issues. In (2A), we fine-tune a language model on $\langle \text{original code}, \text{type} \rangle$ dataset to learn how to estimate the issue type from source code changes performed by developers. In (2B), we fine-tune a language model on $\langle \text{original code}, \text{revised code} \rangle$ dataset to predict parts of the code that need to be revised. The *revised code* variable refers to the modified version of *original code* after taking into account the comments of reviewers. Finally, we use our two recommender systems in (3) as an assistant to the code review process. By predicting the type and location of issue from the code changes made by developers, we can better guide the reviewers in the code review process.

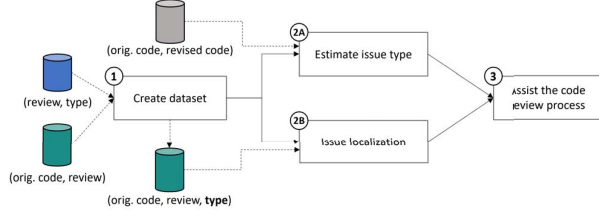


Fig. 2. Proposed approach overview

B. Datasets

In this work, we have used an existing small dataset of reviews' comments¹ labeled with six types of issues: *Testing*, *Error/ Resource handling*, *Naming*, *Documentation*, *Code Organization/ Refactoring* and *Other*. We used this dataset to train our models in (1). Table I explains each of these types and illustrates it with a sample from the dataset.

Additionally, we have retrieved a large dataset of pull requests for 19 popular projects using *Github REST API*. The list of 19 projects is taken from [33]. Each project has several pull requests where developers request to merge their modifications with the main branch and make these changes available to others. Each pull request has review comments related to specific parts of the source code.

We extracted 119,994 pull requests from 19 GitHub repositories, with a total of 342,742 reviews (i.e., comments and source code). The retrieved dataset is composed of triplets $\langle \text{original code}, \text{review}, \text{revised code} \rangle$ where *review* is a comment made by the reviewer on the *original code*. We use the component (1) to augment it with the type of issue. This dataset is used in (2A) and (2B).

C. Ensemble learning-based approach to predict the issue type from review comment (1)

In this section, we go through the details of our first contribution. As shown in Figure 3, our proposed framework consists of an ensemble learning-based model (i.e., meta-learner (C)), that combines a learning-based expert (A) with a knowledge-based expert (B), to predict the type of issues addressed in the review comments. We will use this component to label our retrieved dataset with the types of issues.

¹<https://zenodo.org/record/7533156>

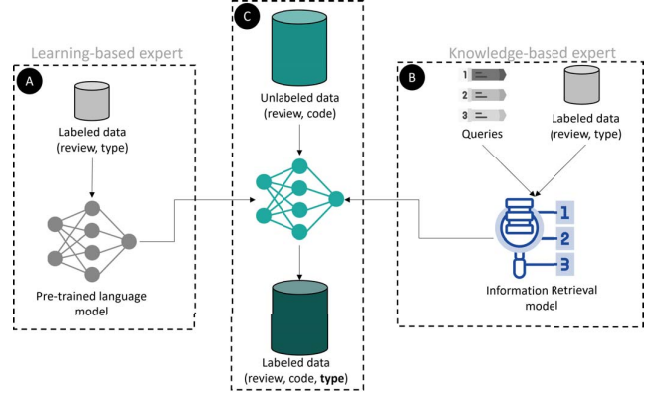


Fig. 3. Predict issue type from review comment

Our component is composed of three main steps: creating a learning-based expert (A), defining a knowledge-based expert (B), then building a meta-learner to learn how to best combine the two experts predictions (C).

1) Learning-based expert (A)

The goal of this step is to build a deep learning model that predicts the type of issue (among the supported types in Table I) from the comments of reviewers. We fine-tuned RoBERTa, a pre-trained language model, on a dataset of reviews' comments labeled with the types of issues.

RoBERTa uses *Byte-Pair Encoding*, a subword-based tokenization algorithm that breaks unseen words into several sub-words, such that frequently seen sub-words can be represented by the model [34]. The advantage of this algorithm is that it solves problems related to word-based tokenization (e.g., large vocabulary size, out-of-vocabulary problem, etc.) and character-based tokenization (e.g., long sequences, loss of semantics from the individual tokens, etc.).

In addition to tokenization, we removed code fragments from the reviews' comments to prevent the model from overfitting to code tokens, i.e., avoid that the model learns from code tokens, as we aim to predict the issue type from reviewers' comments. Then, we fine-tuned this model on this dataset to produce our learning-based expert.

2) Knowledge-based expert (B)

Due to a lack of data and to increase performance, we define a second information retrieval expert based on our code review knowledge.

Information retrieval (IR) is the activity of finding relevant information, to the user's needs, in a corpus. The user writes a query to convey his needs, then, the IR model returns a ranked list of the most relevant documents, based on their relevance scores, from a large corpus of documents.

For instance, IR models are commonly used in search engines where the corpus is constituted of all the web resources on the world wide web. The query is a set of keywords written by the user in the search bar, by which he explains his intents and needs. The IR model calculates the relevance score of each document in the corpus with respect to the user query. Finally,

TABLE I
THE TYPES OF REVIEWS' COMMENTS USED BY OUR APPROACH WITH ILLUSTRATING SAMPLES

Type	Explanation	Sample Comments
Testing	Comments related to testing.	"It's not used by all tests... some tests need their own mock, so I kept it on a test scope (I think it's clearer)."
Error/ Resource handling	Involves comments related to error, exception and resource handling, bugs, etc.	"We can throw an 'IllegalArgumentException' there."
Naming	Related to comments involving naming/renaming variables, methods, classes, etc.	"Rename 'keyHex' to 'keyHexOrKeyId' to indicate this can support AWS KMS as well."
Documentation	Comments related to documentation, comments, javadocs, licenses, etc.	"Why have the double license header here ?"
Code Organization/ Refactoring	Involves comments connected with refactoring, modularization, code optimization, etc.	"Can we move this 'if' block inside the 'if [...] ; then' block?"
Other	General comments (e.g., no changes are needed) or related to other types like code semantics.	"Such a pull request would be nice."

it returns the list of the most pertinent documents ranked with their relevance score.

We formulate our problem (i.e., predicting the issue type from the review comment) as an information retrieval problem. We define a query q_t for each type $t \in T$ where T denotes all the supported issues types presented in Table I. The dataset of reviews' comments is the corpus, that is denoted as R such that $R = \{r_i\}_{0 \leq i \leq n}$ where r_i is the i^{th} review comment and n denotes the size of the corpus. We build a query q_t for each type of issue t . Given a type t and its associated query q_t , the IR model M returns a ranked list of reviews' comments $L_t = \{r_{t,i}\}_{1 \leq i \leq n}$, such that:

- $i \leq j \Leftrightarrow rel_{t,i} \geq rel_{t,j}$ such that $rel_{t,i} = relevance(r_{t,i})$ denotes the relevance of the i^{th} review r_i with respect to the query q_t
 - $R = L_t$
- a) *Corpus*

The corpus is formed of the reviews' comments. We performed pre-processing of these reviews. We removed code fragments to prevent the model from using code tokens to detect the type (e.g., avoid using the name of a method to predict that the review is related to testing). We performed stemming so that inflected words could be detected when their roots match keywords.

b) Queries

We define a query for each type of issue. Table II shows an excerpt of the defined queries for the different types. Each query is composed of keywords correlated to the corresponding type. We performed stemming on the keywords for homogeneity with the pre-processing of the corpus. This would allow reducing the size of the queries (i.e., not adding derived words, of a keyword, to the query).

c) IR model:

We used BM25 [35], which is a commonly used model in information retrieval thanks to its high performance. Given a query and a document, BM25 computes the relevance score according to the following formula:

$$score(D, Q) = \sum_{i=1}^n IDF(q_i) \cdot \frac{tf(q_i, D) \cdot (k_1 + 1)}{tf(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl}\right)}$$

TABLE II
EXCERPT OF THE DEFINED QUERIES. EACH TYPE HAS A QUERY THAT IS COMPOSED OF CORRELATED KEYWORDS.

Naming	Documentation	Error	Test	Refactoring
name	documentation	exception	test	refactor
rename	doc	error	assert	pull
uppercase	todo	resource	fail	remove
lowercase	comment	throw	pass	move
suffix	header	issue	success	restructure
prefix	description	bug	unit	unused
camelcase	rephrase	fix	cover	split
.....				

where $\left\{ \begin{array}{l} D : \text{document} \\ Q : \text{query containing tokens } q_1..q_n \\ q_i : \text{query token} \\ n : \text{size of the query} \\ tf(t, D) : \text{frequency of term } t \text{ in document } D \\ idf : \text{inverse document frequency} \\ avgdl : \text{average document length in the collection} \\ k_1, b : \text{hyper-parameters} \end{array} \right.$

d) Results:

Given a corpus and a query, an IR model returns the list of documents ranked by their relevance score. In our case, it returns the list of comments ranked with their relevance score with respect to the query.

Figure 4 illustrates how our knowledge-based expert works. The IR model (i.e., BM25) takes as input a corpus of reviews and a set of queries (i.e., one query per type as shown in Table II). Then it returns for each query q_t (corresponding to the type t) a list of ranked reviews L_t . Each review has a relevance score $rel_{t,i}$ with respect to each query. That is, each review r_i has a vector of relevance scores $V_i = [rel_{t,i}]_{t \in T}$. The predicted type of each review is the type where the review has the maximum relevance score for the corresponding query.

This classification process is parameterized by a threshold β . If all of the relevance scores for the different types, normalized by the sum of the scores, are below this threshold, we assume that none of the types is relevant to the review comment. Thus, we assign this comment to the *Other* category.

That means, the predicted type p_i for r_i is defined as follows:

$$p_i = \begin{cases} \text{argmax}(V_i) & \text{if } \frac{\max(V_i)}{\sum(V_i)} \geq \beta \\ \text{'Other'} & \text{otherwise} \end{cases}$$

such that $\beta \in [0, 1]$ and argmax returns the type t that corresponds to the maximum score in V_i .

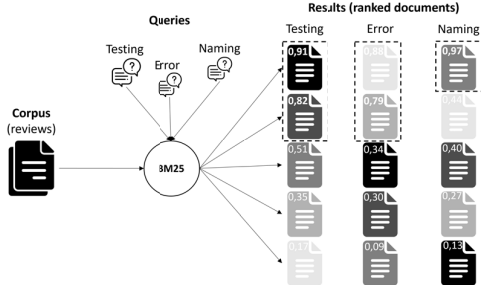


Fig. 4. Knowledge-based expert: an information retrieval model to predict the type of reviews

3) Meta-learner: a stacking-based model to combine the two experts (C)

To produce a more efficient and robust model, we propose to combine the two prior defined experts. To reach this, we use *stacking*, which is an ensemble learning method. It consists of training a *meta-learner* that takes as input the predictions of several models and tries to learn how to combine them to generate the final predictions with better precision.

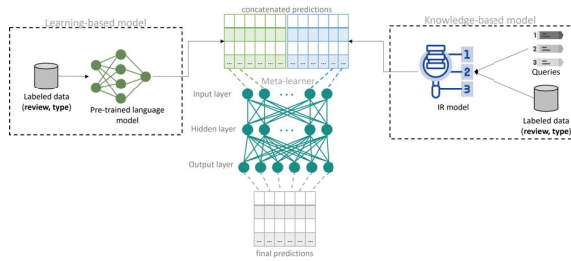


Fig. 5. A stacking-based meta-learner that learns from the concatenated predictions of the two experts to generate the final predictions.

Figure 5 depicts our proposed framework to combine the two experts. For a review comment r_i given as input, $U_i = (u_1, u_2, \dots, u_{|T|})$ is the prediction (i.e., output vector) of the learning-based expert where $|T|$ is the number of reviews and u_j corresponds to the probability of the review r_i to be of type t . $V_i = (v_1, v_2, \dots, v_{|T|-1})$ is the output of the knowledge-based expert where v_j is the relevance score of the review r_i with regards to the query q_j (query corresponding to the issue type t_j). We concatenate these predictions in one vector W_i such that $W_i = [U_i V_i] = [u_1, u_2, \dots, u_{|T|}, v_1, v_2, \dots, v_{|T|-1}]$ that is fed as input to a neural network (with one hidden layer and a softmax output layer) to train it on predicting the final decision about the type of review.

We use this produced model to label our large dataset of $\langle \text{original code}, \text{review}, \text{revised code} \rangle$ triplets. The new dataset is a set of 94,121 quadruplets $\langle \text{original code}, \text{review}, \text{revised code}, \text{type} \rangle$.

D. Fine-tune language models to predict the issue type from source code (2A)

In this part, the goal is to build a deep learning model that can predict the type of issue from the original source code submitted by developers. To achieve this, we fine-tuned some pre-trained language models (CodeBERT, GraphCodeBERT, RoBERTa, and BERT) on this task using our built dataset (i.e., we use $\langle \text{original code}, \text{type} \rangle$ couples as input to our model). Table III shows the data distribution over the labels.

Some pre-trained language models (e.g., CodeBERT) are dedicated to programming languages. Given a fragment of code, these language models generate contextual representations (i.e., embeddings). Some other pre-trained language models (e.g., RoBERTa) are dedicated to natural language. For these, we have performed some pre-processing (e.g., split camel case words) on the source code to extract code tokens. These are fed to the language model to learn how to predict the type of issue from code tokens.

E. Fine-tune language models to locate issues in the source code (2B)

This component is dedicated to locating issues in the source code. We perform the difference between *revised code* and *original code* using *difflib*, a python library that compares two sequences and output the changed lines in the input to generate the output. Then, we flag each line in the *original code*, if it is changed, as problematic. We consider a line that is changed by a developer, after a comment made by the reviewer, as problematic. We end up with 1,174,551 entries in our dataset (i.e., lines of code that have issues). We randomly selected the same number of lines of code, but that has not been changed between the initial and revised version. The resulting dataset is composed of 2,349,102 couples $\langle \text{line of code}, \text{issue} \rangle$, where *issue* is 1 if the line is changed in the revised code (i.e., the line is problematic) and 0 if not. We add some context to each line of code. The resulting dataset is a set of quadruplets $\langle \text{upper context}, \text{current line}, \text{lower context}, \text{issue} \rangle$. *Current line* is the line of code that we are predicting if it has an issue or not. The upper and lower contexts are defined according to a window w . *Upper context* is defined by w lines of code before the current line and *lower context* is defined by w lines of code after the current line. We fine-tune some pre-trained language models to predict if a code line is problematic. Figure 6 illustrates the proposed approach. We have as input a line of code, along with its upper and lower contexts. The fine-tuned model predicts if the corresponding line has an issue.

F. Assist the code review process (3)

The produced models, from (2A) and (2B), are used in to assist reviewers in the code review process. Given a code change submitted by a developer that is impacting one specific file, we

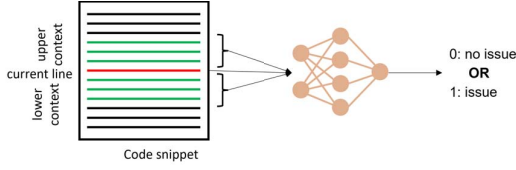


Fig. 6. Approach for issue location: the model takes as input a line of code and some upper and lower contexts, and predicts if the corresponding line has an issue.

use our fine-tuned model to infer the type and location of issue that might exist in the source code. This would give a prior knowledge about the type and location of issue which helps jump-start the reviewer to understand the problem. Having this information at hand, the reviewer can write a description about the identified issue and probably recommend a way to fix it.

In practice, code changes might impact several files and might be related to several issues. To deal with this, we propose to process these changes file-wise. We split code changes per file. For each code change related to one file, we use our models to identify issues if they exist.

IV. EVALUATION

To evaluate the efficiency of our proposed framework, we conducted a set of experiments. In this section, we define the research questions, describe the experimental setting, then present and discuss the results.

A. Research questions

To validate our approach, we define four main research questions. For each of these questions, we use three metrics, precision, recall, and F1 score.

- **RQ1:** *Is our model, learned in the first step of our approach (Figure 2-①), able to correctly predict the issue type given the review comment?*
- **RQ2:** *For the same step, how does the meta-learner perform compared to the learning-based expert and the knowledge-based experts, considered separately?* We compare the performance of our meta-learner with the knowledge-based expert and the learning-based expert as well as their linear combination.
- **RQ3:** *Is our approach able to correctly predict the issue type from the original code, and what is the best performing language model for this task?* We compare the effectiveness of four language models.
- **RQ4:** *Is our approach able to correctly locate the issue in the source code, and what is the best performing language model for this task?* We compare the effectiveness of four language models.

B. Experimental setting

To address the different research questions, we conducted two sets of experiments: a user study for RQ1 and RQ2, and two empirical studies based on large labeled datasets to answer respectively RQ3 and RQ4.

1) User study: assess the meta-learning-based labeling approach

To answer RQ1 and RQ2, we conduct a user study to assess the effectiveness of our approach in predicting the type of issue from the review text. Further, this would allow us to ensure that our model can accurately label our large dataset of $\langle \text{review}, \text{code} \rangle$ pairs.

For the learning-based expert, we fine-tuned RoBERTa for 30 epochs using the Adam optimizer with a learning rate of 0.01 and a weight decay of 0.001. The weight decay is a regularization technique applied to the weights of the model that allows to prevent overfitting and avoid gradient exploding. For the knowledge-based expert, we use BM25 model as an information retrieval model with default parameters $b = 0.75$ and $k1 = 1.2$. These typical values work pretty well for most corpuses. We tested several values of β and we found that $\beta = 0.35$ produces fairly good results. As explained in Section III-C2, we define a query for each issue type (Table II). The two experts are combined using a meta-learner (i.e., ensemble learning), a one-layer neural network with a softmax activation function. We use the produced model to label our large dataset of $\langle \text{review}, \text{code} \rangle$ pairs with issues' types. The resulting dataset contains triplets $\langle \text{review}, \text{code}, \text{type} \rangle$.

To assess the correctness of this labeling, we conduct a user study where we asked participants to evaluate the predictions generated by our proposed framework. To ensure a smooth and easy experience with our experiments, we created and hosted a web application. A participant is provided with a detailed description and reference examples of the experiment and is asked to evaluate as many labels as possible. At each step, the participant is provided with a random review comment and its predicted label and is asked to answer if this label is correct or not. If the predicted label is incorrect, the participant should suggest other labels, among labels presented in Table I, that better correspond to the review comment. Otherwise, the participant can select additional labels if the review comment is about several issue types. Optionally, participants can leave feedback or change another example. Figure 7 shows the web application developed for this user study.

To enroll participants in the study, we sent a call for volunteers requiring knowledge in continuous integration, particularly code review. We followed a convenience sampling method, by sending invitations to research groups in our network. Thirty-one participants responded positively, who are researchers (Ph.D. students, postdoctoral fellows, or professors) with good expertise in code review. We shared, with the participants, a link to the web application. We made it available for one week where people could participate in the experiment anonymously. Participants were shown examples of reviews' comments randomly. However, we perform balancing inside each category of labels to ensure a balanced evaluation.

2) Empirical study: evaluate the effectiveness of our fine-tuned model on predicting reviews' types from source code

To answer RQ3, we used a labeled dataset of 94,116

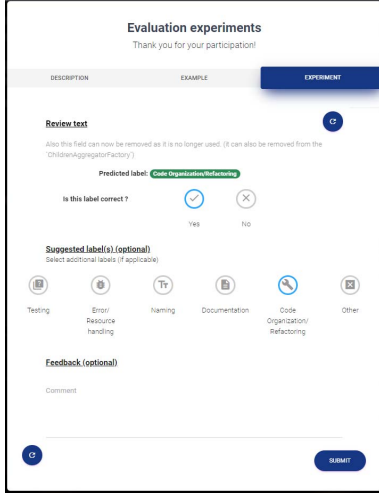


Fig. 7. Web application used in the user study

TABLE III
DISTRIBUTION OF SAMPLES OVER THE LABELS IN THE DATASET

Label	Original dataset	Number of samples		
		Undersampled dataset	Train	Test
Other	46 116	4 878	1 249	6 127
Code organization	13 594	4 869	1 258	6 127
Documentation	11 463	4 911	1 216	6 127
Error handling	10 057	4 880	1 247	6 127
Testing	6 834	4 933	1 194	6 127
Naming	6 127	4 938	1 189	6 127
Total	94 121	29 409	7 353	36 762

samples. Each sample is a pair $\langle \text{review}, \text{code} \rangle$. We used our proposed framework to label each sample using the review comment. The resulting dataset is a set of pairs $\langle \text{code}, \text{type} \rangle$.

Table III shows the distribution of the samples over the types in the dataset. We note that the dataset is imbalanced as there is an unequal distribution of classes within the dataset (e.g., 46,116 samples of type “Other” and only 6,127 of type “Naming”). Training a model on an imbalanced dataset could lead to poor performance as it might cause a bias towards the majority class (i.e., overfitting). To solve this problem, we perform *undersampling* which is a common technique used to balance the dataset. Undersampling consists of keeping all the data in the minority class and decreasing the size of other classes by randomly deleting samples. This allows extracting more accurate information from originally imbalanced datasets.

As shown in Table III, the resulting dataset has 6,127 samples of each class, thus a total size of 36,762. We split the dataset into train set and test set: 80% of the data is used for training and 20% for testing. We fine-tune four language models (i.e., BERT [25], RoBERTa [28], CodeBERT [36], and GraphCodeBERT [37]) using the training set for 30 epochs. We used Adam optimizer with a learning rate of 0.01 and a weight decay of 0.001. Then, we evaluate the fine-

TABLE IV
SUMMARY OF THE USED LANGUAGE MODELS

Model	Description
<i>BERT</i>	Bidirectional Encoder Representations from Transformers (BERT) is a model based on the transformer’s architecture that is pre-trained on a large corpus of raw text with two objectives: Masked language modeling (MLM) and Next sentence prediction (NSP).
<i>RoBERTa</i>	RoBERTa is a model based on the transformers architecture that improves upon BERT. It is pre-trained on a large corpus of raw text using the MLM objective. RoBERTa reduces the computational costs and improves the masking algorithm compared to BERT.
<i>CodeBERT</i>	CodeBERT is a transformer that has exactly the same architecture as RoBERTa. The model is initialized with RoBERTa weights and trained, on bi-modal data (i.e., documents and code) of CodeSearchNet [38], with LM and Replaced Token Detection (RTD) objectives.
<i>GraphCodeBERT</i>	GraphCodeBERT is a pre-trained model for programming languages that considers the inherent structure of code. It is a multi-lingual model that supports several programming languages (e.g., Java, Python, Go, etc.).

tuned language models on the test set by computing the three performance metrics, precision, recall, F1-score. Table IV summarizes the language models used in this experiment.

3) Empirical study: evaluate the effectiveness of our fine-tuned model on predicting issue location in the source code

To answer RQ4, we used a labeled dataset of 2,349,102 couples $\langle \text{line of code}, \text{issue} \rangle$ where *issue* is a boolean variable indicating whether the line of code is problematic or not. The dataset is balanced between the two classes (i.e., 1,174,551 in each class). We split the dataset into 80% for training and 20% for testing. Then, we fine-tuned four language models: BERT, RoBERTa, CodeBERT, and GraphCodeBERT (Table IV). We fine-tune each language model on the training set for 30 epochs. We use Adam optimizer with a learning rate of 0.01 and a weight decay of 0.001. Then, we evaluate each model on the test set by computing the precision, recall and F1-score.

C. Results

1) Results for RQ1

Table V shows the results of the user study to evaluate the effectiveness of our model in predicting the issue type from the review comment. We received 262 responses. Based on these responses, we computed some performance metrics (i.e., precision, recall, and F1-score). The support, of a specific class, is the number of actual occurrences of this class in the dataset. Based on the support scores, we notice that the different groups have close sizes since we have balanced the test set proposed to the participants during the experiments.

Our meta-learner has an average F1 score of 88%. The model has good precision and recall for all the classes. Given this high performance, this model can be used to correctly label the large dataset and use it as ground truth to learn issue

TABLE V
RESULTS OF OUR PROPOSED FRAMEWORK ON PREDICTING THE TYPE OF
ISSUE FROM THE REVIEW COMMENT

Type	Precision	Recall	F1	Support
Other	0.90	0.91	0.90	49
Code organization	0.86	0.82	0.84	38
Documentation	0.90	0.78	0.83	45
Error handling	0.79	0.96	0.87	48
Testing	0.95	0.86	0.90	42
Naming	0.90	0.93	0.91	40
Average	0.88	0.88	0.88	43.67

types from code changes. **We assume that the small margin of error (around 10%) is acceptable and that deep learning models can overcome this bias given that we have a large dataset with diversified examples.**

2) Results for RQ2

A meta-learner model that combines several experts can make better predictions and achieve better performance results than a single expert. To confirm this statement, we define RQ2 to compare the effectiveness of the proposed meta-learner, which combines the two experts, to the learning-based and knowledge-based experts, considered separately. As a baseline, we also compare our meta-learning-based approach to the linear combination of the two experts that is defined as follows:

$$S_L(r, t) = \lambda S_{RoBERTa}(r, t) + (1 - \lambda) S_{BM25}(r, t)$$

where

- r : review comment
- t : issue type
- S_{LCE} : score function of the linear combination of experts
- $S_{RoBERTa}$: RoBERTa score function
- S_{BM25} : BM25 score function
- λ : weight that expresses the importance of each expert

Each score function returns a value that expresses the confidence that a given review r belongs to the class (i.e., issue type) t . λ is a parameter between 0 and 1 that expresses our confidence in the ability of each expert to generate correct predictions.

To answer RQ2, we consider the responses collected from participants as a ground truth. Then, we compute the performance metrics (i.e., precision, recall, F1-score) for each of the experts and their linear combination.

Table VI reports the results of the knowledge-based expert (i.e., BM25), the learning-based expert (i.e., fine-tuned RoBERTa) and for their linear combination. We compare these results to those of the meta-learner proposed in our approach (Table V).

We tested several values of λ on the training set. We set λ to 0.7, as it gave the best results. The average F1 score is 79% for BM25, 83% for RoBERTa, and 85 for the linear combination of the two experts. Our proposed approach had better results as we got 88% as F1 score. The weights of the meta-learner are learned and optimized to have the best results which led to a more effective model compared to other baselines.

3) Results for RQ3

To answer RQ3, we fine-tuned the four language models using the training set (i.e., $\langle \text{original code, type} \rangle$ pairs). Then, we evaluated and compared the effectiveness of the resulting models on a test set of unseen examples of size 7,353, as shown in Table III, using the performance metrics. Table VII summarizes the obtained results.

GraphCodeBERT showed the best accuracy (0.80) and CodeBERT the second-best (0.77) whereas BERT and RoBERTa have worse accuracy (0.56 and 0.57 respectively). BERT and RoBERTa are two language models dedicated to natural language, this explains their overall poor performance, although they have good precision and recall for some labels. However, CodeBERT and GraphCodeBERT are cross-modal pre-trained models for programming languages. This explains the high-performance gap between these and the other models in predicting some labels like “code organization/refactoring” and “Naming”. As these issue types are source code-specific, it is hard to infer them from the code tokens. Other types of issues, like “Error/Resource handling” and “Testing” could be inferred from the code tokens as they have specific keywords (e.g., catch, throw, exception, test).

GraphCodeBERT was the most effective model for this task. Despite its high performance in most of the labels, it was less for the “Other” type. This was expected due to the high variability and diversity inside this class, as it contains several types of issues (e.g., semantics, design, concurrency). This class might also mean the non-existence of issues in the source code.

4) Results for RQ4

To answer RQ4, we fine-tuned the four language models using the training set (i.e., $\langle \text{line of code, issue} \rangle$ pairs). Then, we evaluated and compared the effectiveness of the resulting models on a test set of unseen examples of size 1,174,551 using performance metrics. Table VIII summarizes the obtained results.

GraphCodeBERT outperforms other models, in locating the issue in the source code, with 0.70 as F1-score. CodeBERT has good performance (0.64) whereas other models have worse results (0.53 and 0.57). RoBERTa and BERT are two language models designated for natural language. That is, they are trained on natural language keywords and they do not understand the structure of source code. This makes it hard for these models to perform tasks that require a good understanding of source code.

We investigate the effect of the window size w on the performance of GraphCodeBERT on the issue location task. The window size w denotes the size (i.e., number of lines) of *upper context* and *lower context*. Table IX summarizes our findings.

We note that increasing the window size leads to a better performance. That is, including more context improves the accuracy of our model since this enables it to have more information from the context. Discarding the context (i.e., $w = 0$) leads to a very bad performance. This shows that a single

TABLE VI
RESULTS ON PREDICTING THE TYPE OF ISSUE FROM THE REVIEW COMMENT FOR BM25, ROBERTA AND THEIR LINEAR COMBINATION

Type	BM25			RoBERTa			Linear combination		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Other	0.91	0.72	0.81	0.84	0.88	0.86	0.91	0.88	0.89
Code organization	0.61	0.75	0.67	0.88	0.80	0.84	0.84	0.77	0.80
Documentation	0.67	0.85	0.75	0.72	0.85	0.78	0.78	0.86	0.82
Error handling	0.72	0.84	0.78	0.85	0.86	0.85	0.86	0.85	0.85
Testing	0.87	0.89	0.88	0.86	0.90	0.88	0.92	0.90	0.91
Naming	0.85	0.80	0.83	0.82	0.81	0.81	0.87	0.82	0.84
Average	0.77	0.81	0.79	0.83	0.85	0.83	0.86	0.85	0.85

TABLE VII
RESULTS OF THE LANGUAGE MODELS IN PREDICTING THE ISSUE TYPE FROM THE SOURCE CODE. P REFERS TO PRECISION, R TO RECALL AND F TO F1-SCORE.

Label	BERT			RoBERTa			CodeBERT			GraphCodeBERT		
	P	R	F	P	R	F	P	R	F	P	R	F
Other	0.36	0.37	0.36	0.38	0.40	0.39	0.53	0.54	0.53	0.57	0.60	0.59
Code organization/Refactoring	0.29	0.35	0.32	0.27	0.31	0.29	0.71	0.69	0.70	0.79	0.76	0.77
Documentation	0.64	0.67	0.65	0.63	0.67	0.65	0.83	0.85	0.84	0.84	0.84	0.84
Error/Resource handling	0.79	0.80	0.79	0.78	0.79	0.78	0.87	0.88	0.87	0.92	0.92	0.92
Testing	0.82	0.81	0.81	0.83	0.84	0.83	0.84	0.87	0.85	0.86	0.86	0.86
Naming	0.49	0.42	0.45	0.51	0.47	0.49	0.81	0.82	0.81	0.81	0.80	0.80
Accuracy	0.56			0.57			0.77			0.80		

TABLE VIII
RESULTS OF THE LANGUAGE MODELS IN PREDICTING THE ISSUE LOCATION FROM THE SOURCE CODE. WE USE A WINDOW SIZE $w = 3$.

Label	BERT	RoBERTa	CodeBERT	GraphCodeBERT
Precision	0.55	0.59	0.67	0.71
Recall	0.51	0.56	0.62	0.69
F1-score	0.53	0.57	0.64	0.70

TABLE IX
PERFORMANCE OF GRAPHCODEBERT ON ISSUE LOCATION TASK USING DIFFERENT WINDOW SIZES.

Window size	Precision	Recall	F1-score
$w = 0$	0.29	0.31	0.30
$w = 1$	0.52	0.57	0.54
$w = 3$	0.71	0.69	0.70
$w = 5$	0.73	0.72	0.72

line, often, is not problematic per se, but it can be in a certain context. Then, expanding the window size leads to a significant improvement of the performance. However, this improvement is less important after a certain context size as shown in the transition between $w = 3$ and $w = 5$.

Figure 8 illustrates a use case example of our approach. This example is retrieved from our dataset. We correctly predict that *line 4* has an *Error/Resource handling* issue. The *FileInputStream* resource should be declared inside a *Try-with-resources* statement. It ensures that the resource is closed at the end of the statement execution and thus prevent a resource leak.

Having the issue location and type at hand, the reviewer can write the appropriate comment to describe the issue and

```

1 public static void main(final String[] args) throws IOException {
2     final Properties props = new Properties();
3     if (args != null && args.length == 1) {
4         props.load(new FileInputStream(args[0]));
5     }
6
7     props.putIfAbsent(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount-processor");
8     props.putIfAbsent(StreamsConfig.BootstrapServersConfig, "localhost:9092");

```

Fig. 8. Use case example of our approach: our approach correctly predicts that there is an issue of 'Error/Resource handling' and localizes the issue in line 4

likely recommend a way to fix it. Figure 9 shows the review comment written by the reviewer for this example and the revised version of code pushed by the developer to fix the issue and satisfy the comment.

```

props.load(new FileInputStream(args[0]));

Review: Missing 'try' block to close the 'FileInputStream' properly

try (final FileInputStream fis = new FileInputStream(args[0])) {
    props.load(fis);
}

```

Fig. 9. Example of initial code, review comment, and revised code retrieved from our dataset

D. Threats to validity

The evaluation of the different components of the proposed framework showed the effectiveness of our model in predicting the issue type from the source code changes made by developers, as well as the lines of code that are likely responsible for that issue. However, some threats might limit the validity of the evaluation results presented above.

A first threat involves the nature of the data (i.e., reviews' comments) that are noisy and might contain non-English or

misspelled words. This is mitigated by the use of state-of-the-art language models that use Byte-Pair Encoding, which is a subword-based tokenization algorithm that breaks unseen words into several frequently seen sub-words that can be represented by the model.

A second threat relates mainly to the labeling of the large dataset using the meta-learner (i.e., ensemble learning approach). The accuracy of the proposed meta-learner is 88%. Despite this high performance, there is still noise in the training data. BERT-based models, which are pre-trained on huge amounts of data, help mitigate this issue since they are robust and efficient. BERT-based models have a significant performance in terms of bias mitigation [39]. Another threat related to this is the potential subjectivity of the participants involved in the evaluation of the large-dataset labeling. We addressed this threat in two ways. Firstly, we ensured that the selected participants are advanced programmers who have experience in continuous integration. Secondly, we had a training session in which we explained each issue type and showed examples of the triplets *<review, code, type>*.

The last threat entails the small size of the dataset used in the first step to predict the issue type from the review comment. This may lead to poor generalization and performance. To overcome this, we use a combination of two experts to have a more robust model in terms of performance and generalization. Further, knowledge-based experts do not use a training dataset, they rather rely on the knowledge expressed by experts to make decisions. For the learning-based expert, pre-trained language models could be fine-tuned using small amounts of data while showing a good performance.

V. RELATED WORK

In the literature, many works have investigated the automation of code review using deep learning models. Despite the efforts dedicated to the automation of code review, most of the works have focused on the later stages of this process.

Several approaches have been proposed to learn the relevance between code changes and reviews. These works use a predefined and static dataset of reviews' comments. Based on the code changes made by developers, these approaches recommend the most relevant reviews for this code. In [15], the authors propose a multi-encoder deep learning model, that based on the code changes and their context recommends the most relevant reviews from a predefined dataset. Another work, proposed in [33], introduced a more sophisticated approach based on multi-level embedding to learn the relevancy between code and reviews. This approach uses word-level and character-level embeddings to better represent the semantics provided by code and reviews.

Tufano et al. [16] proposed an approach that partially automates code corrections before and during code review. This approach is composed of two main components: a 1-encoder transformer encoder that recommends additional changes to the developer before submitting his code and a 2-encoders transformer that suggests the needed changes on the source code to satisfy the reviewer comments [16]. The authors

improved their approach, in [17], using T5, a pre-trained text-to-text transfer transformer [29]. This new approach allowed to support non-trivial scenarios of code changes involving the introduction of new identifiers.

Some works tackled the automation of other tasks in the code review process. Shi et al. [40] proposed a model that predicts whether the revised code submitted by a developer should be accepted or not. [41] presented a model that locates parts of the source code impacted by a reviewer's comment. Chouchen et al. [42] introduced a multi-objective approach to recommend the best suitable reviewer to inspect a submitted code change.

Despite their importance, these works do not provide explanations for the type or location of issue that exists in the source code. In fact, it is highly important, for developers, to understand the decisions made by deep learning models. That is, building explainable and interpretable recommender systems will allow users to better trust and understand the decisions made by these systems [43]–[45]. We believe that our work is a step forward in the understanding of the decisions made by the recommender systems, used in the late stages of the code review process (i.e., recommendation of review comments and code changes).

VI. CONCLUSION

We propose an effective framework to predict the type and location of issues from the code changes made by developers. In an initial step, we used ensemble learning to accurately label a large dataset of triplets *<original code, review, revised code>* with the *issue type*. Then, we used state-of-the-art language models to identify the type and localize the issue in the source code. We conducted a user study to evaluate the dataset labeling task. The results show that our labeling is precise enough with a good accuracy of 88%. We conducted two empirical studies to evaluate the performance of our fine-tuned models on localizing the issue and predicting its type. GraphCodeBERT showed the best performance for these tasks. Our framework recommends useful information to assist reviewers during the code review process. We believe that providing the issue type and location in the code to reviewers is more useful for them than writing the review on their behalf. Indeed, having this information, the reviewer could write creative and subtle comments that can be more useful and informative to the developer and potentially pointing to the way to solve it.

As part of future work, we aim to target the problem of issue localization differently. We plan to explore the attention weights of our fine-tuned language models when determining the type of issue from the code. This would allow to determine which code fragments are contributing to the identification of a particular issue and that need to be improved by developers. We intend also to use text-to-text models to suggest templates of review comments to support reviewers in writing the comments. Finally, we plan to evaluate, through a user study, what level of support and automation is the more useful to reviewers.

REFERENCES

- [1] M. Fowler and M. Foemmel, "Continuous integration," <https://martinfowler.com/articles/continuousIntegration.html>, 2006.
- [2] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
- [3] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*. Springer, 2002, pp. 575–607.
- [4] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 81–90.
- [5] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: an effective verification process," *IEEE software*, vol. 6, no. 3, pp. 31–36, 1989.
- [6] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, 1984, pp. 13–40.
- [7] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the Qt, VTK, and ITK projects," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 171–180.
- [8] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.
- [9] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 434–448, 2008.
- [10] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 192–201.
- [11] —, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.
- [12] B. A. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D. W. R. Marsh, "Industrial perspective on static analysis," *Software Engineering Journal*, vol. 10, no. 2, pp. 69–75, 1995.
- [13] P. Bielik, V. Raychev, and M. Vechev, "Learning a static analyzer from data," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 233–253.
- [14] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 598–608.
- [15] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.
- [16] R. Tufan, L. Pascarella, M. Tufanoy, D. Poshyanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 163–174.
- [17] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyanyk, and G. Bavota, "Using pre-trained models to boost code review automation," *arXiv preprint arXiv:2201.06850*, 2022.
- [18] U. Lichtenthaler, "Substitute or synthesis: the interplay between human and artificial intelligence," *Research-technology management*, vol. 61, no. 5, pp. 12–14, 2018.
- [19] —, "Extremes of acceptance: employee attitudes toward artificial intelligence," *Journal of Business Strategy*, 2019.
- [20] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.
- [21] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (Dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- [22] M. C. O. Silva, M. T. Valente, and R. Terra, "Does technical debt lead to the rejection of pull requests?" *arXiv preprint arXiv:1604.01450*, 2016.
- [23] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.
- [24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
- [25] 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.
- [26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [27] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.
- [28] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
- [29] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv preprint arXiv:1910.10683*, 2019.
- [30] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and systems magazine*, vol. 6, no. 3, pp. 21–45, 2006.
- [31] J. Vanschoren, "Meta-learning," in *Automated Machine Learning*. Springer, Cham, 2019, pp. 35–61.
- [32] R. Vilalta and Y. Drissi, "A perspective view and survey of meta-learning," *Artificial intelligence review*, vol. 18, no. 2, pp. 77–95, 2002.
- [33] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.
- [34] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.
- [35] S. Robertson, H. Zaragoza, and M. Taylor, "Simple bm25 extension to multiple weighted fields," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, 2004, pp. 42–49.
- [36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
- [37] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.
- [38] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
- [39] V. S. S. Settipalli and N. M. K. Dasireddy, "Reducing unintended bias in text classification using multitask learning," Master's thesis, Faculty of Computing, Blekinge Institute of Technology, 2021.
- [40] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 4910–4917.
- [41] V. J. Hellendoorn, J. Tsay, M. Mukherjee, and M. Hirzel, "Towards automating code review at scale," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1479–1482.
- [42] M. Chouchen, A. Ouni, M. W. Mkaouer, R. G. Kula, and K. Inoue, "Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review," *Applied Soft Computing*, vol. 100, p. 106908, 2021.
- [43] J. E. Mercado, M. A. Rupp, J. Y. Chen, M. J. Barnes, D. Barber, and K. Procci, "Intelligent agent transparency in human-agent teaming for multi-uxv management," *Human factors*, vol. 58, no. 3, pp. 401–415, 2016.
- [44] B. Hayes and J. A. Shah, "Improving robot controller transparency through autonomous policy explanation," in *2017 12th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 2017, pp. 303–312.
- [45] J. Y. Chen, K. Procci, M. Boyce, J. Wright, A. Garcia, and M. Barnes, "Situation awareness-based agent transparency," U.S. Army Research Laboratory, Aberdeen Proving Ground, Tech. Rep. ARL-TR-6905, 2014.