SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

May 2025, Vol. 68, Iss. 5, 152104:1–152104:16
https://doi.org/10.1007/s11432-022-3953-4

E-PRedictor: an approach for early prediction of pull request acceptance

Kexing CHEN[1], Lingfeng BAO[1*], Xing HU[1], Xin XIA[2] & Xiaohu YANG[1]

[1]*State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310058, China*
[2]*Software Engineering Application Technology Lab, Huawei, Hangzhou 310056, China*

Received 27 December 2022/Revised 10 May 2023/Accepted 7 October 2023/Published online 16 January 2025

Abstract A pull request (PR) is an event in Git where a contributor asks project maintainers to review code he/she wants to merge into a project. The PR mechanism greatly improves the efficiency of distributed software development in the open-source community. Nevertheless, the massive number of PRs in an open-source software (OSS) project increases the workload of developers. To reduce the burden on developers, many previous studies have investigated factors that affect the chance of PRs getting accepted and built prediction models based on these factors. However, most prediction models are built on the data after PRs are submitted for a while (e.g., comments on PRs), making them not useful in practice. Because integrators still need to spend a large amount of effort on inspecting PRs. In this study, we propose an approach named E-PRedictor (earlier PR predictor) to predict whether a PR will be merged when it is created. E-PRedictor combines three dimensions of manual statistic features (i.e., contributor profile, specific pull request, and project profile) and deep semantic features generated by BERT models based on the description and code changes of PRs. To evaluate the performance of E-PRedictor, we collect 475192 PRs from 49 popular open-source projects on GitHub. The experiment results show that our proposed approach can effectively predict whether a PR will be merged or not. E-PRedictor outperforms the baseline models (e.g., Random Forest and VDCNN) built on manual features significantly. In terms of F1@Merge, F1@Reject, and AUC (area under the receiver operating characteristic curve), the performance of E-PRedictor is 90.1%, 60.5%, and 85.4%, respectively.

Keywords pull request, prediction model, GitHub

Citation Chen K X, Bao L F, Hu X, et al. E-PRedictor: an approach for early prediction of pull request acceptance. Sci China Inf Sci, 2025, 68(5): 152104, https://doi.org/10.1007/s11432-022-3953-4

1 Introduction

The pull-based development model, a mechanism for collaboration in distributed software development, has been widely used in open-source software (OSS) development. In this model, external developers (also known as contributors) fork a base repository of a project and independently make their code changes. Once the code change is ready, contributors can submit a pull request (PR) to the main repository. Then, integrators and other developers can review the submitted code changes in the pull request and provide some comments. Finally, the project maintainers decide to accept or reject the PR based on its quality and the discussion among developers.

The pull-based development model is well implemented by GitHub, which is one of the most popular OSS hosting websites. GitHub reported that they had 170 million merged PRs in 2021[1)]. However, since reviewing pull requests requires a significant amount of effort from integrators [1], it has been a heavy burden for integrators when the volume of pull requests increases. For example, Gousios et al. [2] conducted a survey with 645 top OSS contributors and found that the asynchrony characteristic of the pull-based model hinders the observability of the overall status of a project and burdens integrators. On the other hand, the heavy workload of integrators delays the decision on PRs. We find that the average time from PR creation to closure in our collected data was more than 37 days (see Table 1).

To reduce the effort of integrators, many researchers try to build prediction models to predict whether a PR will be merged or not. They extract different aspects of features from PRs and other related information, such as code changes in PRs [3, 4], comments on PRs [5, 6], and developer experience [7].

* Corresponding author (email: lingfengbao@zju.edu.cn)
1) https://octoverse.github.com/.

Table 1 Statistics of top-20 popular projects.

Project name	Stars	PRs	Commits	Merged PRs	Rejected PRs	Merge ratio	Life time	Program language
flutter/flutter	128761	30150	130300	21947	8203	0.73	5 days, 8 h	Dart
nodejs/node	81599	25113	59511	21228	3885	0.85	20 days, 4 h	JavaScript
kubernetes/kubernetes	80718	64954	107130	50000	14954	0.77	27 days, 2 h	GO
angular/angular	76300	19200	38363	3964	15236	0.21	31 days, 12 h	TypeScript
mrdoob/three.js	74056	11664	30134	9194	2470	0.79	31 days, 8 h	JavaScript
puppeteer/puppeteer	73159	2421	5367	1998	423	0.83	7 days, 21 h	TypeScript
vercel/next.js	72941	9006	34605	7309	1697	0.81	8 days, 22 h	JavaScript
tensorflow/models	71175	3320	10941	2192	1128	0.66	80 days, 0 h	Python
mui-org/material-ui	70842	14060	43106	11645	2415	0.83	4 days, 6 h	JavaScript
PanJiaChen/vue-element-admin	70507	457	1625	324	133	0.71	10 days, 4 h	Vue
laravel/laravel	66450	4024	10829	1623	2401	0.40	7 days, 17h	PHP
storybookjs/storybook	64522	7330	26088	5948	1382	0.81	9 days, 9h	TypeScript
nvbn/thefuck	63800	558	1365	471	87	0.84	36 days, 11h	Python
moby/moby	61066	17300	27956	13921	3,379	0.80	15 days, 8h	GO
gothinkster/realworld	60123	227	302	184	43	0.81	41 days, 14 h	Shell
django/django	59499	14639	28361	9906	4733	0.68	29 days, 18 h	Python
apple/swift	57168	38937	85932	34786	4151	0.89	14 days, 3 h	C++
spring-projects/spring-boot	57034	4810	8799	3410	1400	0.71	20 days, 4 h	Java
bitcoin/bitcoin	56858	15283	37883	10965	4318	0.72	36 days, 0 h	C++
pallets/flask	56514	1966	3662	1274	692	0.65	30 days, 17 h	Python
All projects' mean	58147	9698	25695	7346	2352	0.69	37 days, 5 h	–

Gousios et al. [3] performed an exploratory study with 291 projects and found that whether the PR modifies recently modified code is the most important factor that affects the decision to merge a PR. Dey and Mockus [7] built a Random Forest model based on 14 features to predict whether a PR will be merged and achieve a promising performance (i.e., 0.95 in terms of AUC (area under the receiver operating characteristic curve)). However, the prediction models in most of these prior studies are built on the data after PRs are submitted for a while, such as comments on PRs. We think such prediction models are not helpful in practice because integrators still need to inspect the PR and communicate with contributors after a PR is created. The workload of integrators will not decrease. One possible solution is to predict whether a PR will be merged or rejected when it is created. Thus, integrators can receive immediate feedback, then estimate their efforts and prioritize PRs they are working on.

In this study, we take a new direction in predicting whether a PR will be merged or rejected when it is created. Thus, the prediction results can help integrators estimate the effort of pull requests and prioritize them immediately. On the other hand, contributors can also receive quick feedback and take action to improve their pull requests immediately. We proposed E-PRedictor to predict whether a PR will be merged using the information when it is created. First, similar to the previous studies [7, 8], E-PRedictor extracts three dimensions of manual features, including developer profile, specific pull request, and project profile. Also, it leverages deep semantic features by using BERT models [9] to encode a PR's description and code changes. Finally, we use XGBoost [10] to build a prediction model by combining both manual and deep semantic features.

We collected 475192 PRs from the most popular OSS project on the GitHub to evaluate the performance of E-PRedictor. We also chose some classical classifiers (i.e., Logistic Regression, Decision Tree, Random Forest, and XGBoost) and a deep neural network classifier (i.e., VDCNN) built on the manual features as the baselines. The experiment results showed that E-PRedictor can effectively predict whether a PR will be merged or rejected when created in terms of F1@Merge, F1@Reject, and AUC. The F1-scores of E-PRedictor for merged and rejected PRs are 90.1% and 60.5%, respectively. E-PRedictor outperforms the baselines by a statistically significant margin. We also evaluate the importance of each dimension of features in E-PRedictor and the effectiveness of E-PRedictor in the cross-project setting.

Our paper makes the following contributions.

(1) Based on previous work and manual inception, we proposed an updated PR merge detection approach. We build a dataset containing 475192 PRs from 49 popular OSS projects. We provide a replication package of our dataset and the proposed approach, which is available at https://github.com/

ckxkexing/pr-acceptance.

(2) We propose a prediction model named E-PRedictor, which can predict whether a PR will be merged or rejected when it is created. E-PRedictor is built on 46 manual features from three dimensions (contributor profile, specific pull request, and project profile) and 30-dimensional deep semantic features based on the description and code changes of the PR.

(3) We evaluate E-PRedictor on our collected PRs. The experiment results show that E-PRedictor can effectively predict the acceptance of PRs in terms of multiple metrics (e.g., F1@Merge, AUC). E-PRedictor outperforms several baselines (e.g., XGBoost built on manual features) significantly. We also find that the deep semantic features from the PR description are the most important dimension in E-PRedictor.

Paper organization. Section 2 introduces the concepts of prediction models used in our study. Section 3 shows the process of the data collection and analysis. Section 4 describes our approach. Sections 5 and 6 present our experiment setup and results, respectively. Section 7 discusses the implications and threats to validity of our work. Section 8 presents related work. Section 9 concludes the paper and discusses the future work.

2 Background

In this section, we will introduce the background knowledge of PR early characteristics, pre-trained models, and classification models used in our study.

2.1 Pull request early characteristics

Gousios and Zaidman [11] first presented a heuristic to determine whether a PR on GitHub has been merged. He also categorized the features of PRs into three dimensions: PR-specific, project-specific, and developer-specific. The PR-specific characteristics include the change in the number of lines and the characteristics of the files modified. The project-specific characteristics include the number of lines of code and the number of project members. The developer-specific characteristics include the number of PRs and the number of followers the developer has before creating a new PR. In total, there are 27 features, out of which 21 are related to when the PR is created.

Dey and Mockus [7] mined pull request in NPM repository. The pull request features were classified into five categories, comprising a total of 50 features, namely PR creator, specific PR, NPM package repository, specific head repository, and base repository. After performing random tree cross-validation, 14 valid features were identified, out of which 10 were related to early PR features. On their randomly split 7:3 dataset, they achieved an area under curve-receiver operating characteristics (AUC-ROC) of 0.77 and an accuracy of 0.72.

Zhang et al. [12] expanded on Gousios and Zaidman's [11] work and proposed a total of 95 features across the three types of PR features, out of which 33 were relevant at the time of PR creation.

We summarize these features that are not used in our approach as follows.

At the developer level, we do not use the features that are a combination of the other features. For example, prior_interaction [12] is based on the number of issues events, PR events, commits, etc., which have been included in our features. Another similar feature is social_strength [12]. Additionally, although several features are missing, some features in our study are highly correlated with them. For example, core_member [12] indicates whether a developer is a project committer, or has permission to merge and close PRs. We believe that the features in the developer profile dimension of our study can indicate the expertise of a developer, which is similar to core_member [12] and prior_review_num [12].

At the PR level, we do not use the number of added files [11], deleted files [11], and modified files [11]. But these features are very similar to the number of committed touch files we use. Moreover, some features in these previous studies are project specific or not suitable for the projects in our dataset, such as intro_branch [7], and the lines of code (LOCs) of test files [11].

At the project level, we use prs_cnt_in_month and issue_cnt_in_month instead of open_issue_num [12] and open_pr_num [12]. Additionally, Zhang et al. reported that sloc, test_lines_per_kloc, pushed_delta, and project_age are not the factors that explain pull request decisions the most. So we do not include these in our datasets.

2.2 BERT family

BERT (bidirectional encoder representations from transformers) [9] has gained great attention in natural language processing tasks. Compared with other pre-training model structures, BERT considers the two-way relationship between each word in the input text. It uses the encoder structure of Transformers to train the two tasks of mask language model (MLM) and next sentence prediction (NSP) on the English Wikipedia dataset. It achieves state of the art on 11 downstream tasks. Experiments have proven that BERT has outstanding ability in word understanding and context understanding.

RoBERTa [13] is a pre-training model with robustness optimization based on BERT. Thus, we use the RoBERTa model instead of the BERT model to encode the description of pull requests. Its training dataset size is ten times that of the original BERT and uses larger byte-pair encoding (BPE) [14]. It uses a dynamic MASK mechanism to replace the static MASK mechanism for model training, which improves the shortcoming that the BERT model needs a duplicate dataset in training. Moreover, increasing the trained batch size and training epoch on this model is conducive to improving the performance of the model.

CodeBERT [15] has the same structure as RoBERTa, but it is a multilingual programming model trained by natural language and programming language. It uses MLM and replaced token detection (RTD) [16] as the objective function on a dataset containing six programming languages (Python, Java, JavaScript, PHP, Ruby, and Go). The RTD objective function requires the model to predict the original placement of each input word, making model pre-training more efficient. Experiments have found that the CodeBERT model has better results in natural language code search and code document generation tasks. We use the CodeBert model to encode the code changes in pull requests.

2.3 Predict models

We choose several classical prediction models, including Logistic Regression, Decision Tree, Random Forest, and XGBoost in our study, which have been used in the previous studies that investigate whether a PR will be merged [7, 12, 17, 18]. In addition, we also used a deep neural network (i.e., VDCNN) since deep learning techniques have shown promising performance in many software engineering studies.

Decision Tree is a tree model used for prediction or regression tasks. The internal nodes of the tree represent the judgment of a feature, each branch represents the output of a judgment result, and finally, each leaf node represents a classification result. Decision Tree is created from top to bottom on the training data set, and each node split is based on the change of entropy. All prediction nodes are in the same tree, and the closer the internal nodes are to the root node, the greater the impact on the prediction result. Therefore, it is easier to find out the key features in the trained Decision Tree.

Random Forest [19] is an ensemble learning method that uses multiple Decision Trees to perform prediction tasks together. The model will randomly select some data for training a Decision Tree, and this Decision Tree only randomly selects some features for the calculation of entropy changes in node splitting. Finally, a large number of different Decision Trees trained by this method constitute Random Forests. Random Forest randomly selects features and training data during training, which alleviates the problem of over-fitting and is conducive to improving the prediction performance.

XGBoost [10] is a kind of tree boosting model. Its principle is to integrate many weak classifier models to form a strong classifier. XGBoost algorithm uses a new cart regression tree model to fit the residual results of the previous tree so that the sum of the predicted value of each tree is close to the ground truth values. As a result, the prediction result is the weighted sum of the value of the corresponding node of the input in each tree.

VDCNN [20] is a very deep convolutional network applied to text processing. It uses a VGG-like or ResNet-like architecture to go deeper because, in the field of computer vision, deep convolutional networks like ResNet have excellent classification capabilities. The model uses text characters as input and calculates the results through stacked convolutional blocks. Tests on different data sets show that VDCNN using 29 convolutional blocks can achieve better results.

3 Data collection and analysis

In our study, we focus on popular engineered software projects [21]. Specifically, we used GitHub Restful API to collect the top-100 popular projects regarding the number of stars (in September 2021). Then

we filtered the projects using the following criteria: (1) non engineered software projects, such as the collection of study materials and tutorials, which are not within the scope of our study; (2) the forked projects; (3) the main code development activities of the project are not carried out on the GitHub platform, which only serves an open-source role on GitHub. For example, Google/Guava is Google's internal repository which shares source code and PRs in GitHub masked by copybar-service[2)]. So we removed Google/Guava and Keras-team/Keras. To apply the first filter, we removed the projects with the following keywords in their descriptions: "awesome", "tutorial", "education", etc. Finally, we select the top 49 projects from the result list to get a controllable number of PRs. Compared with the previous study that only used NPM projects [7], we believe that our collected projects are diverse.

For each project, we used data from GHTorrent [22] in our study. Specially, we analyzed GHTorrent's MySQL database snapshot of 2021–03–06. Beside PR data, we could get other development activities from GHTorrent, including commits, issues, and discussions among developers. In the obtained PR data, we removed the PRs that have not been closed since their status is not eventually determined. We determined whether a PR was merged using the following heuristics.

(1) If the field "merged_at" (indicating the merged time for a PR) of the PR from GitHub Restful API is not null.

(2) If one commit of the PR is included in the project.

(3) The last three comments in the PR can be matched by the keywords indicating the merge action (e.g., "merged", "landed", "pushed", "included", and "committed") and contain a commit identifier. Sometimes a pull request cannot be applied correctly, or when project-related policies do not permit automatic merging sometimes [3]. For example, the code change of a PR[3)] in the project nodejs/node is included in another PR, which is mentioned in the comments. In our dataset, 6066 PRs (about 1.3% in our datasets) were identified as merged through the use of this checker approach. We randomly selected and manually verified 20 PRs that were identified as merged from each project. It is important to note that we only examined projects with more than 20 such PRs. We find that the detection accuracy was found to be 75.8%. Given that only 1.3% of our dataset consists of this type of merged PRs, we believe that these discrepancies have minimal impact on our study.

(4) The PR is mentioned by a commit in the repository. Because sometimes integrators can create a patch copied from the PR and commit it to the repository [3].

For the remaining PRs, we regarded them as rejected.

A PR contains two parts: the textual description and a set of commits. For the textual description of PR, we removed the noisy information in the body content, such as tags and special characters (e.g., '#') in the markdown files. For the commits in the PR, we extracted the file names and the corresponding code changes. We only kept the source code files and discarded other types of files (e.g., configuration files). For the code changes, we only kept the added and deleted lines.

Table 1 presents the statistics of the selected projects in our study. Due to the page limitation, we only list the top 20 popular projects in Table 1. As shown in Table 1, except for a few projects (e.g., PanJiaChen/vue-element-admin), most projects have a large number of PRs. The average number of PRs among the selected projects is 9698. The mean percentage of merged PRs in these projects is 0.69, which indicates that our dataset is imbalanced. But there are several projects whose merged ratio is low, such as angular/angular. This might be because such projects have high criteria to accept a PR in the main branch. Among the selected projects, JavaScript is the most used programming language (16 projects), followed by Python (8 projects). Moreover, we compute the time interval between the creation time and the closure time of a PR. On average, the life time of PRs in our dataset is about 37 days, which might indicate a heavy workload of integrators.

4 Approach

In this section, we introduce the process of E-PRedictor, which is shown in Figure 1. First, E-PRedictor extracts three dimensions of manual features (i.e, contributor profile, specific pull request, and project profile) based on the development activities in GitHub. Second, it uses BERT models to convert the description and code changes of PRs into encoding features. Finally, we build an XGBoost classification model based on the manual features and deep semantic features.

2) https://github.com/apps/copybara-service.

3) https://github.com/nodejs/node/pull/28048.

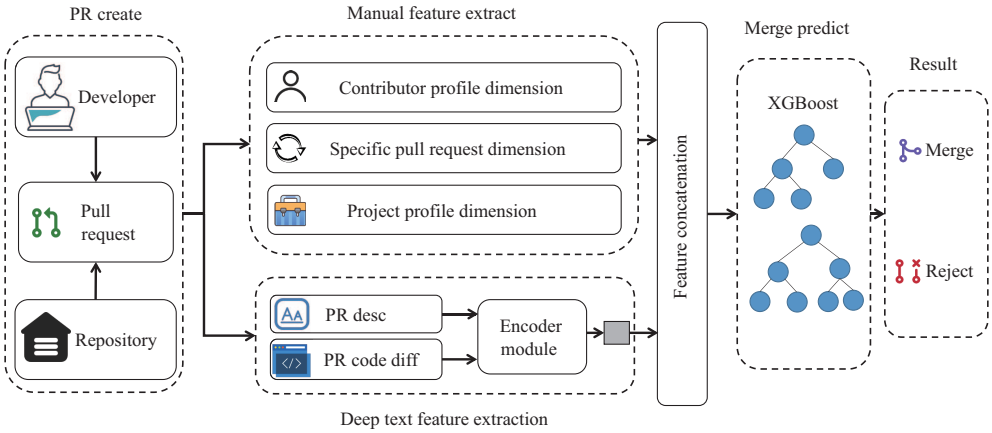

Figure 1 (Color online) Overall framework of E-PRedictor.

Table 2 Manual features used in E-PRedictor.

Dimension	Feature name	Description
Contributor profile	before_pr_user_projects	Number of projects owned by the contributor before the submitted PR
	before_pr_user_commits	Number of the contributor's commits before the submitted PR
	before_pr_user_pulls	Number of the contributor's PR submitted before the submitted PR
	before_pr_user_issues	Number of the contributor's issues submitted before the submitted PR
	before_pr_user_followers	Number of the contributor's followers before the submitted PR
	before_pr_user_commits_proj	Number of projects the contributor had contributed before the submitted PR
	bot_user	Whether it is a robot account
	issue_created_in_project_by_pr_author	Number of issue created in the project before the submitted PR
	issue_created_in_project_by_pr_author_in_month	Number of issue created in the project within one month before the submitted PR
	issue_joined_in_project_by_pr_author	Number of issue joined in the project before the submitted PR
	issue_joined_in_project_by_pr_author_in_month	Number of issue joined in the project within one month before the submitted PR
	number_of_created_pr_in_this_proj_before_pr	Number of created PR in the project before the submit created by the contributor
	number_of_merged_pr_in_this_proj_before_pr	Number of merged PR in the project before the submit created by the contributor
	number_of_closed_pr_in_this_proj_before_pr	Number of closed PR in the project before the submit created by the contributor
	ratio_of_merged_pr_in_this_proj_before_pr	Ratio of merged PR in the project before the submit created by the contributor
Specific pull request	is_this_proj_first	Is it the first PR of the contributor for the project
	pr_desc_len	Length of the description of the submitted PR
	check_pr_desc_mean	Is the length of the description of the PR greater than the average length of created PRs
	check_pr_desc_medium	Is the length of the description of the PR greater than the medium length of created PRs
	pr_commit_count	Number of commits in the submitted PR
	commit_add_line (sum/max/min)	Lines of the added code in the commits of the submitted PR
	commit_delete_line (sum/max/min)	Lines of the deleted codes in the commits of the submitted PR
	commit_total_line (sum/max/min)	Lines of the added and deleted lines in the commits of the submitted PR
	commit_file_change	Number of changed files in the commits of the submitted PR
	whether_pr_created_before_commit	Whether a commit already exists when the PR is created
	contain_test_file	Whether the change contains a test file
	contain_doc_file	Whether the change includes a document file
Project profile	before_pr_merge_cnt	Number of merged PR contributors had contributed before the submit
	before_pr_closed_cnt	Number of closed PR contributors had contributed before the submit
	before_pr_merge_ratio	Ratio of merged PR contributors had contributed before the submit
	before_pr_project_commits	Number of commits in the project before the submitted PR
	before_pr_project_commits_in_month	Number of commits of the project within one month before the submitted PR
	before_pr_project_prs	Number of PRs in the project before the submitted PR
	before_pr_project_prs_in_month	Number of PRs of the project within one month before the submitted PR
	before_pr_project_issue	Number of issues in the project before the submitted PR
	before_pr_project_issue_in_month	Number of issues of the project within one month before the submitted PR
	before_pr_project_comments_in_pr	Number of comments on all the PRs in the project before the submitted PR
	before_pr_project_comments_in_pr_in_month	Number of comments on PRs in the project within one month before the submitted PR
	before_pr_project_issues_comment	Number of comments on all the issues in the project before the submitted PR
	before_pr_project_issues_comment_in_month	Number of comments on issues in the project within one month before the submitted PR

4.1 Manual feature extraction

In this section, we follow the previous studies in software analytics [7, 8] to extract different dimensions of features to build prediction models, including contributor profile dimension, specific pull request dimension, and project profile dimension. Table 2 shows the details of these features, which are as follows.

Contributor profile dimension. This dimension refers to features extracted from the overall information of a contributor when he/she creates a pull request, which is dependent on his/her development activities in GitHub. Similar to the study of Dey and Mockus [7], we use six features to measure a contributor's profile, which are as follows. The before_pr_user_project feature quantifies the number of projects

owned by the contributor. The before_pr_user_commits, before_pr_user_pulls, before_pr_user_issues features quantify the number of commits/pull requests/issues submitted by the contributor before he/she creates a new pull request, respectively, which indicates his/her history activities in GitHub. The before_pr_user_followers feature quantifies the number of developers who follow the contributor, indicating the programming and social ability of the contributor. The before_pr_user_commits_proj feature is an indicator of the contribution that the developer contributed to the project before. These features might indicate the professional experience and ability of a contributor in the open source community. The bot_user is based on the user's login name to differentiate between fewer bot accounts that perform regular tasks. In addition, issue-related features indicate the developer's enthusiasm for participating in open source. And contributors' past PR submissions on the same project indicate their experience level. An experienced developer might be more likely to be familiar with the requirements of a project and submit high-quality pull requests. Therefore, we believe that these features might affect a pull request to be merged or rejected.

Specific pull request dimension. This dimension refers to features computed based on the information of the submitted pull request, which might indicate the quality of a pull request. This dimension has been adopted in previous studies [7, 23]. We collect several types of information for a pull request. First, we check whether it is the first time for the contributor to submit a pull request in the project (is_this_proj_first). Many OSS projects have specific requirements (e.g., using a pull request template) for submitting pull requests. A developer's first pull request in the project might be rejected since he/she is not familiar with such requirements. Second, we compute some metrics based on the description of pull requests because a low-quality description of pull requests might hinder the understanding and readability of pull requests for integrators. For example, a pull request with a short sentence might be rejected due to a lack of enough information. We calculate the length of the description of a pull request (pr_desc_len), and determine whether the length of the description is greater than the mean and median length of all pull requests in a project (check_pr_desc_mean and check_pr_desc_median), respectively. Third, we calculate several metrics based on the commits in the pull requests. The code change in a pull request is the most important factor that affects whether it is to be merged or rejected. We count the number of commits in a pull request (pr_commit_count), the number of added lines (commit_add_line), deleted lines (commit_delete_line), and the total changed lines (commit_total_line) in commits. Also, we count the number of changed files in commits of a pull request (commit_file_change). Some PRs are created earlier than the first commit. We use the weather_pr_created_before_commit to record it. Inspired by Yue et al. [24], whether PR modifies documents and tests files (contain_test_file and contain_doc_file) are also checked.

Project profile dimension. This dimension refers to features extracted from the overall information of a project when a contributor creates his/her pull request, which is similar to the macro-climate of a project (e.g., workload) used in previous studies [8, 25]. In Adam's research [26], different projects have different governance styles, which affects the acceptance rate of PRs. We count the number of merged PRs (before_pr_merge_cnt) and closed PRs (before_pr_closed_cnt), and calculate merge ratio (before_pr_merge_ratio). Dey and Mockus [7] also reported that the number of pull requests that existed prior to the submission of a pull request has an impact on its accepted results. We count the number of commits, issues, and pull requests in the project before the specific pull request is created (before_pr_project_commits, before_pr_project_issues, and before_pr_project_prs). We also count the number of comments on pull requests (before_pr_project_comments_in_pr) and issues (before_pr_project_issues_comments) in the project because the discussion between project maintainers and contributors is an indicator of the activeness and openness of a project. Additionally, we also count these metrics within one month before the specific pull request is created (e.g., before_pr_project_commits_in_month). Because we think that recent development activities are good indicators of the working environment in an OSS project [8].

4.2 Deep textual feature extraction

Figure 2 presents the structure of the encoder module in E-PRedictor. For the description of PRs, we use RoBERTa to generate encoding features since RoBERTa is a pre-trained model with robustness optimization based on BERT. For the code changes of PRs, we use CodeBERT to generate encoding features because CodeBERT has shown better performance on some tasks related to the source code than the original BERT. We use RoBERTa and CodeBERT to map the description and code changes of

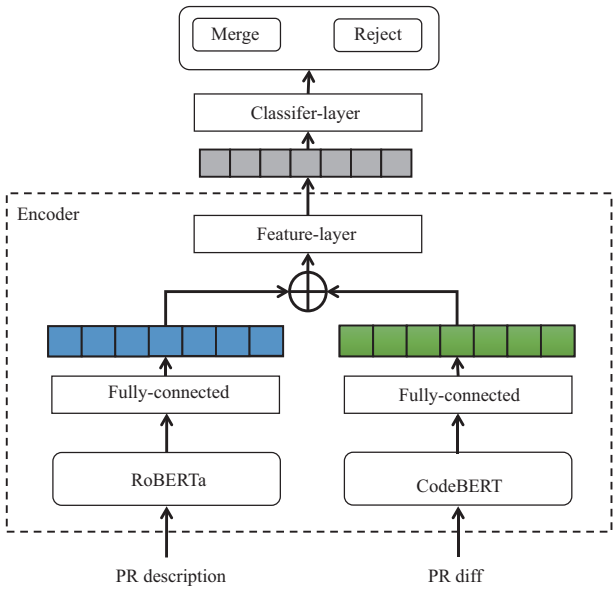

Figure 2 (Color online) Encoder module.

PRs into a deep semantic vector space with a simple fine-tuning process.

To fine-tune the BERT models, we simply use the labels of PRs (merged or rejected) as the training targets. The description and code changes of PRs through the encoder module will be encoded into two 768-dimensional encoding vectors, which is too high for the classical classifier used in E-PRedictor. Hence, the two vectors will be spliced together and passed through a fully connected layer to generate a 30-dimensional vector. Then, we pass the encoded vector through an auxiliary fully connected layer to get a two-dimensional vector for calculating the loss with the ground truth. We use the BCE (binary crossentropy) loss function, which is used for binary classification tasks. Finally, given the description and code change of a PR, we use the encoder module to generate 30-dimensional encode features.

4.3 PR acceptance prediction

After extracting manual features and deep semantic features, we obtain a total of 76 features, including 46 manual features and 30 deep semantic features. Then, we build an XGBoost model based on these features. We use the default parameters for all XGBoost models in our study.

5 Experiment setup

In this section, we describe the baselines and evaluation metrics used in our study. We evaluate our proposed approach on the collected dataset, containing 475192 PR from 49 OSS projects. We run the experiment on an Ubuntu 20.4 LTS machine with an Intel (R) Xeon (R) CPU with 16 cores and an Nvidia GeForce RTX 3090 graphics card with 24 gigabytes of memory.

5.1 Baselines

Many previous studies have used traditional classifiers to predict the chance of acceptance for PRs, such as Random Forest [7] and Logistic Regression [12]. These traditional classification models based on multiple dimensions of manual features have achieved promising performance in the prediction of PR acceptance. Therefore, in this study, we choose four classical classifiers (i.e., Logistic Regression, Decision Tree, Random Forest, and XGBoost) based on the manually extracted features as the baseline models. Furthermore, there are several deep encoding techniques for textual data and source code. To verify the effectiveness of the encoder module in E-PRedictor, we use the VDCNN model instead of the BERT models in E-PRedictor. We use the default training optimizer for VDCNN, i.e., SGD (stochastic gradient descent optimizer), with a learning rate of 0.01 and momentum of 0.9.

5.2 Evaluation metrics

For each pull request, there would be four possible outcomes: a merged pull request can be classified as merged (TP) or rejected (FP), while a rejected pull request can be classified as rejected (TN) or merged (FN). Based on these possible outcomes, we calculate precision, recall, and F1-score for each label to evaluate the performance of E-PRedictor and baseline models. We also use AUC (area under the receiver operating characteristic curve) to evaluate the effectiveness of the proposed prediction models.

Accuracy is the number of correctly classified PRs (both merged and rejected) over the total number of PRs, i.e., $\frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$.

Precision@Merge is the proportion of PRs that are correctly labeled as merged among those labeled as merged PRs, i.e., $\frac{\text{TP}}{\text{TP+FP}}$.

Recall@Merge is the proportion of merged PRs that are correctly labeled, i.e., $\frac{\text{TP}}{\text{TP+FN}}$.

Precision@Reject is the proportion of PRs that are correctly labeled as rejected among those labeled as rejected PRs, i.e., $\frac{\text{TN}}{\text{TN+FN}}$.

Recall@Reject is the proportion of rejected PRs that are correctly labeled, i.e., $\frac{\text{TN}}{\text{TN+FP}}$.

F1-score is the summary measure that combines precision and recall, i.e., the harmonic mean of precision and recall. We compute F1-scores for each label, which is widely used in many software engineering studies [7, 8, 17, 27–30].

AUC is an independent threshold measure, which is different from threshold-dependent measures (e.g., F1-score) that often rely on a probability threshold (e.g., 0.5). The value of AUC ranges from 0 to 1. The higher an AUC value, the better the performance of a classifier. Previous studies consider an AUC value of 0.7 as a promising performance score [31, 32].

6 Experiment results

In this section, we present the experiment results by answering the following research questions.

RQ1: Can E-PRedictor effectively predict whether a PR would be merged?

Approach. First, we used 10-fold cross-validation to evaluate the results of the prediction models used in the study. In 10-fold cross-validation, we randomly divide the dataset into ten folds. Of these ten folds, nine folds are used to train the classifier, while the remaining one fold is used to evaluate the performance. The class distribution in the training and testing datasets is kept the same as the original dataset to simulate real-life usage of the algorithm. Second, we also use time-aware validation to evaluate these prediction models, which is also used in the previous studies [8, 27, 28]. In this setting, PRs are sorted in chronological order of the created time and then divided into 10 non-overlapping windows of equal sizes. We use the first n windows as the training data and the $(n+1)$-th windows as the testing data (n from 1 to 9). The reported performance of the models is the average of the results of all runs for both validation settings. We also apply Wilcoxon signed-rank test [33] with Bonferroni correction [34] to investigate whether the improvements of E-PRedictor over the baselines are statistically significant. We used the default configuration of `scipy.stats.wilcoxon`, which is paired, two-tailed with 0.05 Alpha. And we use Cliff's delta [35], which is a non-parametric effect size, to measure the amount of difference between the performance of E-PRedictor and the baselines.

For BERT models, we fine-tune them on the training set with the input word length of 128 and the batch size of 100. We use AdamW provided by Hugging Face [36] as the optimizer and set the initial learning rate to 1E−5.

Results. Tables 3 and 4 present the performance of E-PRedictor and baseline models in terms of the metrics used in this study for 10-fold cross-validation and time-aware validation, respectively. As shown in Tables 3 and 4, E-PRedictor achieves the best performance in terms of all metrics. In terms of AUC, the performance of E-PRedictor is promising, i.e., 85.4% and 81.6% for 10-fold and time-aware validation settings, respectively. The F1@Merge of E-PRedictor is 90.1% and 89.3% for 10-fold cross-validation and time-aware validation, respectively. However, in terms of F1@Reject, the performance of E-PRedictor is not high (i.e., 60.5% and 53.4%). This is because our data set is slightly imbalanced (the ratio of merged PRs to rejected PRs is approximately 7:3, see Table 1). But the Precision@Reject of E-PRedictor is 75.1% in 10-fold cross-validation, which is acceptable.

Among the classical classifiers, Logistic Regression achieves the worst performance, while XGBoost has the best performance in terms of all metrics. We also find that the performance of the prediction model

Table 3 Results of each prediction model using 10-fold validation.

Models	Accuracy (%)	Precision @Merge (%)	Recall @Merge (%)	F1@Merge (%)	Precision @Reject (%)	Recall @Reject (%)	F1@Reject (%)	AUC (%)
Logistic Regression	76.0	76.0	100.0	86.4	61.4	0.10	0.20	55.7
Decision Tree	79.5	83.5	91.0	87.1	60.2	43.3	50.3	74.0
Random Forest	80.6	85.4	89.8	87.6	61.5	51.4	56.0	79.5
XGBoost	82.4	83.8	95.3	89.2	73.6	41.9	53.4	82.5
VDCNN	82.4	83.8	95.2	89.1	73.2	41.9	53.3	82.5
E-PRedictor	84.1	85.8	94.7	90.1	75.1	50.6	60.5	85.4

Table 4 Results of each prediction model using time-aware validation.

Models	Accuracy (%)	Precision @Merge (%)	Recall @Merge (%)	F1@Merge (%)	Precision @Reject (%)	Recall @Reject (%)	F1@Reject (%)	AUC (%)
Logistic Regression	76.7	77.0	99.3	86.7	65.9	1.40	2.20	55.6
Decision Tree	74.5	81.9	85.9	83.8	44.5	36.7	40.0	64.1
Random Forest	74.6	82.7	84.6	83.6	45.5	41.3	43.1	68.6
XGBoost	80.6	83.0	94.0	88.1	65.7	35.8	46.0	78.3
VDCNN	81.2	83.2	94.6	88.5	67.4	36.3	47.0	78.8
E-PRedictor	82.6	84.8	94.2	89.3	69.5	43.6	53.4	81.6

Table 5 Results of six variants of E-PRedictor using time-aware validation.

Models	Accuracy (%)	Precision @Merge (%)	Recall @Merge (%)	F1@Merge (%)	Precision @Reject (%)	Recall @Reject (%)	F1@Reject (%)	AUC (%)
Hide contributor profile dimension	81.8	84.0	94.3	88.8	67.5	39.9	50.0	78.4
Hide specific pull request dimension	81.9	84.4	93.8	88.8	67.0	42.2	51.7	80.4
Hide project profile dimension	82.1	85.1	93.0	88.8	66.7	45.6	54.0	81.2
Hide PR description	81.3	83.2	94.6	88.6	67.4	36.4	47.1	78.8
Hide PR code change	82.6	84.8	94.3	89.3	69.6	43.5	53.3	81.5
E-PRedictor	82.6	84.8	94.2	89.2	69.5	43.6	53.4	81.6

based on the deep features generated by VDCNN and manual features is similar to that of XGBoost based on only manual features. For example, the F1@Merge of these two models is similar (89.2% vs. 89.1%). Therefore, we believe that the deep semantic features extracted by E-PRedictor are better than those of VDCNN in predicting the acceptance of PRs.

We also find that all the adjusted p-values are smaller than 0.05, which indicates the improvement of E-PRedictor over baselines is statistically significant at the confidence level of 95%, and of large effect size.

E-PRedictor can effectively predict whether a PR would be merged or rejected based on the manual features and deep semantic features.

RQ2: How important is each dimension of features used by E-PRedictor?

Approach. In this study, E-PRedictor uses manual features and deep semantic features to build prediction models. Manual features consist of three dimensions, i.e., contributor profile, specific pull request, and project profile. Deep semantic features come from both the description and the code change of PRs. Thus, in this research question, we want to investigate the importance of each kind of feature in E-PRedictor.

We build six variants of E-PRedictor by hiding one dimension of features. We use 10-fold cross-validation to evaluate the performance of six variants of E-PRedictor. We also apply the Wilcoxon signed-rank test with Bonferroni correction to measure whether the improvement of E-PRedictor over these variants is statistically significant.

Results. Table 5 presents the results of six E-PRedictor variants obtained through 10-fold cross-validation. As shown in Table 5, the original E-PRedictor outperforms these six variants in terms of all metrics. This indicates that the information on each feature dimension contributes to the prediction of PR acceptance. The improvement of the original E-PRedictor on the variants is statically significant (i.e., all p-values are less than 0.05) and at least of small effect size, except the variants which hide PR code change or hide project profile dimension got negligible effect size. This may indicate that the different PRs have less in common at the code level and project level. So less experience can be gained from historical information.

The PR description is the most discriminative dimension, with the F1@Reject of 47.1%. This indicates that the quality of the description of PRs might have an important impact on their acceptance. In terms of AUC, the variant that hides the manual features in the contributor profile dimension achieves the worst performance, i.e., 78.4%. For the other dimensions, the variants have similar performance in terms of all metrics.

The deep semantic features from PR description and the manual features in the contributor profile dimension are the two most important dimensions. However, using all dimensions of features is better.

Table 6 Results of cross-project setting of E-PRedictor.

Projects	F1@Merge (%)		F1@Reject (%)		AUC (%)	
	XGBoost	Ours	XGBoost	Ours	XGBoost	Ours
flutter/flutter	85.0	85.7	61.1	47.0	80.5	73.6
nodejs/node	90.9	90.5	13.3	20.0	68.1	69.6
kubernetes/kubernetes	86.3	77.9	29.0	47.2	66.4	72.5
angular/angular	46.6	41.8	71.2	57.2	73.1	74.5
mrdoob/three.js	87.8	87.1	35.2	44.8	76.8	78.7
puppeteer/puppeteer	90.0	90.1	21.5	40.8	74.9	77.6
vercel/next.js	88.7	88.8	17.5	43.3	72.4	78.1
tensorflow/models	80.0	80.4	45.5	44.8	75.9	75.8
mui-org/material-ui	90.6	89.1	27.2	32.7	74.0	76.1
PanJiaChen/vue-element-admin	82.8	78.9	56.9	67.0	80.6	86.2
Mean	82.9	81.0	37.8	44.5	74.3	76.3

RQ3: How effective is E-PRedictor in a cross-project setting?

Approach. The experiment results in RQ1 and RQ2 show the effectiveness of E-PRedictor on the PRs in our collected dataset. However, a new project may not have enough related data for building a model. Hence, in this RQ, we want to explore the generalizability of E-PRedictor on predicting whether a PR will be merged or rejected in a cross-project setting. For each project, we build a prediction model of E-PRedictor based on the PRs of the remaining projects and use the PRs of this project as the testing data. We also use XGBoost built on the manual features as the baselines because XGBoost achieves the best performance among the classical classifiers.

Results. Table 6 presents the results of F1@Merge, F1@Reject, and AUC of E-PRedictor and XGBoost in the cross-project setting. Due to the page limitation, we only show the results of the prediction models for the top-10 popular projects in our dataset (see Table 1). As shown in Table 6, the performance of E-PRedictor is less to that of XGBoost in terms of F1@Merge (81.0% vs. 82.9%). E-PRedictor has better performance than XGBoost for some projects (e.g., flutter/flutter), while E-PRedictor cannot outperform XGBoost for some projects (e.g., nodejs/node). In terms of F1@Reject and AUC, E-PRedictor outperforms XGBoost for most projects. However, the average values of F1@Reject of E-PRedictor and XGBoost are low, i.e., 37.8% and 44.5%, respectively. This indicates that predicting whether a PR will be rejected in the cross-project setting is difficult. The average AUC of E-PRedictor is 76.3%, which is a promising performance score [31, 32]. Besides, the improvement of E-PRedictor over XGBoost on F1@Reject is statically significant. Overall, E-PRedictor outperforms XGBoost built on the manual features, which indicates that the deep semantic features play an important role in the prediction of PR acceptance in the cross-project setting.

E-PRedictor can effectively predict the acceptance of PRs in the cross-project setting in terms of F1@Merge and AUC. However, its performance in predicting whether a PR will be rejected is not very good.

RQ4: How effective is E-PRedictor built on historical data of a single project?

Approach. In this RQ, we want to explore whether the prediction models built on historical data for a specific project can effectively predict the acceptance of future PRs. We use the time-aware validation used in RQ1 to evaluate the results of the prediction models for a single project. We also use XGBoost built on manual features as the baseline and study the importance features of the input in it.

Results. Table 7 presents the results of F1@Merge, F1@Reject, and AUC of E-PRedictor and XGBoost in the single-project setting for top-10 popular projects. In terms of F1@Merge and AUC, E-PRedictor outperforms XGBoost for most projects and achieves a promising performance (i.e., 76.9% of F1@Merge and 69.4% of AUC). However, the improvement of E-PRedictor over XGBoost on F1@Merge, F1@Reject, and AUC is minor. In other words, interpretable manual features play a large predictive role in this RQ. Hence, we explore the importance of manual features with XGBoost.

We found that the contributors' previous merge experience in that project was generally of high importance. However, as illustrated in Figure 3, the y-axis indicates the feature importance, while the x-axis corresponds to the number of folds for time-aware ten-fold training for various projects. On different projects, we also found different patterns of dynamic of these features' importance in the time dimension. The importance of ratio_of_merged_pr_in_this_proj_before_pr in project flutter/flutter is increasing, but is decreasing in project angular/angular. It changes more smoothly in nodejs/node, but fluctuates more

Table 7 Results of single project setting using time-aware validation.

Projects	F1@Merge (%)		F1@Reject (%)		AUC (%)	
	XGBoost	Ours	XGBoost	Ours	XGBoost	Ours
flutter/flutter	83.2	83.4	40.6	42.5	74.7	75.5
nodejs/node	88.1	89.1	18.1	17.9	65.1	65.7
kubernetes/kubernetes	85.4	86.2	22.4	31.2	66.3	69.4
angular/angular	27.4	29.2	85.3	83.3	60.9	63.8
mrdoob/three.js	88.2	88.2	28.7	27.7	73.5	73.6
puppeteer/puppeteer	88.9	88.5	29.1	33.4	69.6	72.6
vercel/next.js	88.3	87.5	19.6	21.3	67.7	67.7
tensorflow/models	75.9	75.1	44.6	37.4	68.9	69.6
mui-org/material-ui	88.2	88.2	27.3	27.3	68.1	68.1
laravel/laravel	51.1	53.8	67.2	65.4	65.4	68.4
Mean	76.5	76.9	38.3	38.7	68.0	69.4

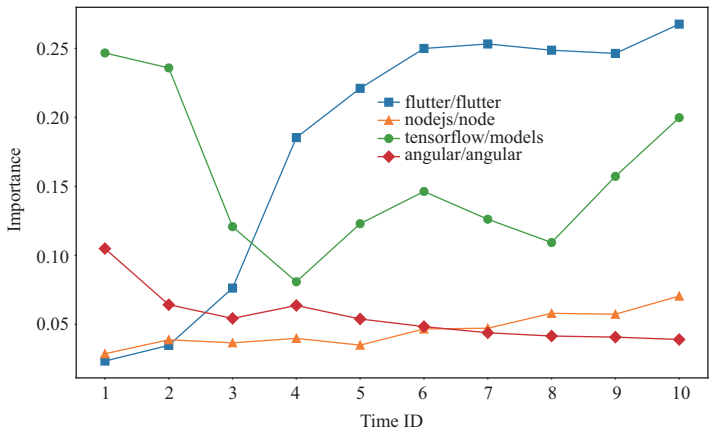

Figure 3 (Color online) Trends of importance in XGBoost of "ratio of merged PR in the project before the submit created".

in the tensorflow/models. These patterns may reflect changes in the demographics of developers in the project. The changes in the importance of experiences in development cooperation may reflect changes in the difficulty of OSS Newcomer participating. Therefore, based on these interpretable manual features, our method can also quantify the status of the project for maintainers.

E-PRedictor built on historical data from a single project can effectively predict the acceptance of its future PRs. Also, the manual features in it are of great help to prediction. In particular, a developer's experience that merged PR within the same project plays an important role in predicting. And the dynamic pattern of manual features' importance can be indicators of project status.

7 Discussion

7.1 Implications

We have the following implications based on the findings in the study.

Combining deep semantic features and manual features is helpful to predict whether a PR can be merged or rejected when it is created. Most prediction models of previous studies are built on the data after a PR is created for a while and achieves excellent performance. For example, the AUC of the Random Forest in the study of Dey and Mockus [7] is 0.95. Compared to these previous studies, our objective is to predict the acceptance of PRs when created and propose E-PRedictor that combines the manual features and deep semantic features. Although E-PRedictor cannot achieve such high performance, its performance is still promising (i.e., 85.4% of AUC in 10-fold cross-validation, see RQ1). We also find that each feature dimension in E-PRedictor contributes to the prediction of PR acceptance (RQ2). There are many other factors affecting the change of PRs being accepted. For example, there are 50 variables mentioned by Dey and Mockus [7]. In the future, we will extract more features to enhance E-PRedictor.

The quality of PR description plays an important role in the acceptance of PRs. In RQ2, we find that the deep features of the PR description are an important dimension in E-PRedictor. We think that the PR description gives the first impression to integrators, and its quality has a significant impact on the acceptance of PRs. Many OSS projects on GitHub have adopted PR templates to help contributors write a good PR description with essential information [4)]. Thus, we suggest that contributors write a clear and high-quality PR description.

There is still room for improvement in predicting whether PR will be rejected. The experiment results show that the F1@Reject of E-PRedictor is not high, e.g., only 60.5% in 10-fold cross-validation. The reason might be the imbalance of data (the ratio of merged and rejected PR is approximately 7:3). However, the precision of E-PRedictor in predicting rejected PRs is acceptable (i.e., 75.1% in 10-fold cross-validation). Thus, we think E-PRedictor can still help integrators save much effort in practice since the rate of true positives on rejected PRs is high. Some technologies (e.g., SMOTE [37]) for imbalanced data can be used to improve the performance of E-PRedictor in predicting whether a PR will be rejected. Furthermore, we need to investigate whether there are some special factors affecting the rejection of PRs to enhance our approach.

Furthermore, our method is a useful auxiliary tool for different participants in open source communities: For researchers, we propose a modified merge judgment method. The PR of most projects will be merged by GitHub merge, and commit merge is also a common form. However, there are PRs that merge through comments, which also need the attention of researchers. In predicting the rejected PR performance, compared with only using the historical information of a single project (38.3% vs. 38.7% in Table 7), the PR description information across projects can improve the performance (37.8% vs. 44.5% in Table 6). This may be that the historical data of the same project are not abundant, while the cross-project data are abundant, and there may be similar unreasonable PR content that is helpful to detect rejected PR. For PR authors, if the model indicates that a PR can be merged, it could encourage PR authors to persist rather than give up easily. Additionally, the model can also expedite reminders for spam PRs created by new contributors and prevent disruption for project administrators. The historical merge rate is a crucial factor in motivating PR authors to persevere, as highlighted by the feature enhancement. Because the characteristics of a developer's past experience are one of the important parts of predicting the PR result. For OSS newcomers, it may be possible to choose a PR with a simple task to practice step by step. And OSS newcomer can imitate the merged PR description content to create PR. For PR reviewers, some PRs may only be newcomer tests or spam PRs, which is particularly common in popular projects. Perhaps through our model method, we can directly close these PRs without directly disturbing the project integrators in the form of emails. In addition, if there is a misjudgment, the PR creator can also reopen the PR.

7.2 Threats to validity

Threats to internal validity refer to errors in our code and experiment bias. We use default settings to pre-train the Bert models and classifiers used in our study. Moreover, to mitigate the bias of results selection, we run 10-fold cross-validation and time-aware validation to evaluate the performance of E-PRedictor and baselines and report the average performance. We focus on changes in feature importance to reduce the impact of different meanings of importance in XGBoost and reality. We also double-check our code; however, there may be some errors that we do not notice. Another threat to internal validity is the criteria that we determine whether a PR is merged or rejected. Since PRs can be merged in different ways [3], we consider three heuristics to search the merged PRs by mining the PRs in a project. However, we still might miss some PRs since the heuristics we use might not cover all merged PRs.

Threats to external validity refer to the generalizability of our findings. The PRs in our study are from 49 OSS projects with the most stars. These projects cover different categories (e.g., front-end, framework) and use different programming languages. The number of PRs in our study is similar to previous studies [7]. Thus, we believe that these projects are representative. In the future, we plan to collect PRs from more OSS projects.

Threats to construct validity refer to the suitability of our evaluation measures. We use accuracy, F1-score, and AUC, which are widely used by prior studies to evaluate the effectiveness of prediction models

4) Creating a pull request template for your repository. https://docs.github.com/en/communities/using-templates-to-encourage- useful-issues-and-pull-requests/creating-a-pull-request-template-for-your-repository. Accessed: Apirl 8, 2022.

in software engineering studies [7, 8, 27, 28]. Thus, we believe that there is little threat to construct validity.

8 Related work

In recent years, there are many studies that analyze pull requests in different ways, such as PR assignment and prioritization, the exploratory studies of the PR quality and PR acceptance.

For example, Veen et al. [38] proposed an approach named PRioritizer to prioritize multiple concurrent PRs by considering multiple factors (e.g., the size of code change, the existence of test cases). Gousios et al. explored both the perspective of the PR creators [39] and the PR integrators [2] by conducting large-scale surveys. They pointed out the challenges and practices in PR creation and merging scenarios. There are some studies that focus on recommending appropriate reviewers for pull requests [40–42].

Many researchers also investigated factors that affect the chance of PRs getting accepted and built prediction models based on these factors. Tsay et al. [5] collected PRs from GitHub Archive and analyzed pull requests in different aspects, including the code changes in PR, the contributors and their social network, and the characteristics of the project. They found that social and technical ability both affect the acceptance of PRs. Gousios et al. [3] found that the decision to merge a pull request is mainly influenced by whether the pull request modifies recently modified code by analyzing 291 projects. Yu et al. [43] collected the information of the continuous integration (CI) tools to analyze the content of PRs from 40 projects in three dimensions of features, i.e., project, pull request, and contributor. They find that the author's experience has a positive impact on PR, while the number of comments contained in PR has a negative impact on PR. Dey and Mockus [7] used PRs in NMP projects to analyze the factors affecting the acceptance of PRs. They built the Random Forest model based on 14 features from multiple dimensions, including the individual aspects of the contributor, the PR's own information, and the characteristics of the NPM projects. Jiang et al. [17] proposed an approach named CTCPPre to predict a PR will be merged. CTCPPre builds an XGBoost model based on the features of code, text, contributor, and project. Zhang et al. [12, 44] summarized the features used in previous studies on PR acceptance prediction and used mixed-effect Logistic Regression models to explain the impact of features on the final state of PR. The PR acceptance in some specific projects is also investigated, such as Linux kernel [45], Firefox [46], Apache [46], and Active Merchant (a commercial project developed by Shopify Inc.) [6].

Compared to these previous articles that study the acceptance of PRs, our proposed approach not only considers different dimensions of manual features including contributor, pull request, and project, but also extracts deep semantic features using BERT models. Additionally, the prediction models of these previous studies are built on the data after a PR is submitted for a while. Our approach performs the prediction of PR acceptance when PRs are initially created, which is more useful in practice.

9 Conclusion and future work

Previous work on predicting the acceptance of PRs often works at the time after PRs are created for a while, which cannot be applied in practice. In this paper, we proposed an approach named E-PRedictor that can predict the acceptance of PRs at their creation time. E-PRedictor combines three dimensions of manual features and deep semantic features by encoding the description and code change of PRs. Then, it builds an XGBoost model based on the extracted features. We collected 475192 PRs from 49 OSS projects to evaluate E-PRedictor. The experiment results show that E-PRedictor effectively predicts whether a PR will be merged or rejected at the creation time. In future work, we plan to collect more PRs from more OSS projects. We also want to collect more information related to PRs and try different deep neural networks to build prediction models. We believe that rich description content and code data can train a more general and robust E-PRedictor model. As a future direction, it would be worthwhile to explore methods such as feature engineering to identify the optimal feature combination from the three feature groups in order to predict the acceptance of PR with minimal manual features. Moreover, by mining the correlation between the features when the PR is created and the dynamic features when the PR is opening, better prediction results of the PR should be obtained.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 62372398, 62141222, U20A20173), National Key Research and Development Program of China (Grant No. 2021YFB2701102), and Fundamental Research Funds for the Central Universities (Grant No. 226-2022-00064).

References

1 Rigby P C, German D M, Cowen L, et al. Peer review on open-source software projects: parameters, statistical models, and theory. ACM Trans Softw Eng Methodol, 2014, 23: 1–33
2 Gousios G, Zaidman A, Storey M A, et al. Work practices and challenges in pull-based development: the integrator's perspective. In: Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), 2015. 358–368
3 Gousios G, Pinzger M, Deursen A V. An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), 2014. 345–355
4 Pooput P, Muenchaisri P. Finding impact factors for rejection of pull requests on GitHub. In: Proceedings of the VII International Conference on Network, Communication and Computing, 2018. 70–76
5 Tsay J, Dabbish L, Herbsleb J. Influence of social and technical factors for evaluating contribution in GitHub. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), 2014. 356–366
6 Kononenko O, Rose T, Baysal O, et al. Studying pull request merges: a case study of Shopify's active merchant. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP), 2018. 124–133
7 Dey T, Mockus A. Which pull requests get accepted and why? A study of popular NPM packages. 2020. ArXiv:2003.01153
8 Bao L, Xia X, Lo D, et al. A large scale study of long-time contributor prediction for GitHub projects. IEEE Trans Software Eng, 2019, 47: 1277–1298
9 Devlin J, Chang M W, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding. 2018. ArXiv:1810.04805
10 Chen T, Guestrin C. XGBoost: a scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016. 785–794
11 Gousios G, Zaidman A. A dataset for pull-based development research. In: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014. 368–371
12 Zhang X, Yu Y, Gousios G, et al. Pull request decision explained: an empirical overview. 2021. ArXiv:2105.13970
13 Liu Y, Ott M, Goyal N, et al. RoBERTa: a robustly optimized BERT pretraining approach. 2019. ArXiv:1907.11692
14 Sennrich R, Haddow B, Birch A. Neural machine translation of rare words with subword units. 2015. ArXiv:1508.07909
15 Feng Z, Guo D, Tang D, et al. CodeBERT: a pre-trained model for programming and natural languages. 2020. ArXiv:2002.08155
16 Clark K, Luong M T, Le Q V, et al. ELECTRA: pre-training text encoders as discriminators rather than generators. 2020. ArXiv:2003.10555
17 Jiang J, Zheng J, Yang Y, et al. CTCPPre: a prediction method for accepted pull requests in GitHub. J Cent South Univ, 2020, 27: 449–468
18 Dey T, Mockus A. Effect of technical and social factors on pull request quality for the NPM ecosystem. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2020. 1–11
19 Breiman L. Random forests. Machine Learn, 2001, 45: 5–32
20 Conneau A, Schwenk H, Barrault L, et al. Very deep convolutional networks for text classification. 2016. ArXiv:1606.01781
21 Munaiah N, Kroh S, Cabrey C, et al. Curating GitHub for engineered software projects. Empir Software Eng, 2017, 22: 3219–3253
22 Gousios G. The GHTorent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), 2013. 233–236
23 Soares D M, de Lima Júnior M L, Murta L, et al. Acceptance factors of pull requests in open-source projects. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC), 2015. 1541–1546
24 Yue Y, Wang Y, Redmiles D. Off to a good start: dynamic contribution patterns and technical success in an OSS newcomer's early career. IEEE Trans Software Eng, 2023, 49: 529–548
25 Zhou M, Mockus A. Who will stay in the FLOSS community? Modeling participant's initial behavior. IEEE Trans Software Eng, 2015, 41: 82–99
26 Alami A, Pardo R, Cohn M L, et al. Pull request governance in open source communities. IEEE Trans Software Eng, 2022, 48: 4838–4856
27 Huang Q, Xia X, Lo D. Supervised vs unsupervised models: a holistic look at effort-aware just-in-time defect prediction. In: Proc eedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017. 159–170
28 Ni C, Xia X, Lo D, et al. Just-in-time defect prediction on javascript projects: a replication study. ACM Trans Softw Eng Methodol, 2022, 31: 1–38
29 Russell R, Kim L, Hamilton L, et al. Automated vulnerability detection in source code using deep representation learning. In: Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 2018. 757–762

30 Alon U, Brody S, Levy O, et al. code2seq: generating sequences from structured representations of code. 2018. ArXiv:1808.01400

31 Lessmann S, Baesens B, Mues C, et al. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Software Eng, 2008, 34: 485–496

32 Nam J, Kim S. CLAMI: defect prediction on unlabeled datasets. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. 452–463

33 Wilcoxon F. Individual comparisons by ranking methods. In: Breakthroughs in Statistics. New York: Springer, 1992. 196–202

34 Abdi H. The Bonferroni and šidák corrections for multiple comparisons. Encyclopedia of Measurement and Statistics, 2007, 3: 103–107

35 Cliff N. Ordinal Methods for Behavioral Data Analysis. New York: Psychology Press, 2014

36 Wolf T, Debut L, Sanh V, et al. Transformers: state-of-the-art natural language processing. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2020. 38–45

37 Chawla N V, Bowyer K W, Hall L O, et al. SMOTE: synthetic minority over-sampling technique. J Artif Intell Res, 2002, 16: 321–357

38 van der Veen E, Gousios G, Zaidman A. Automatically prioritizing pull requests. In: Proceedings of IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), 2015. 357–361

39 Gousios G, Storey M A, Bacchelli A. Work practices and challenges in pull-based development: the contributor's perspective. In: Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016. 285–296

40 Jiang J, Yang Y, He J, et al. Who should comment on this pull request? Analyzing attributes for more accurate commenter recommendation in pull-based development. Inf Software Tech, 2017, 84: 48–62

41 Yu Y, Wang H, Yin G, et al. Reviewer recommender of pull-requests in GitHub. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014. 609–612

42 Yu Y, Wang H, Yin G, et al. Reviewer recommendation for pull-requests in GitHub: what can we learn from code review and bug assignment? Inf Software Tech, 2016, 74: 204–218

43 Yu Y, Wang H, Filkov V, et al. Wait for it: determinants of pull request evaluation latency on GitHub. In: Proceedings of IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), 2015. 367–371

44 Zhang X, Rastogi A, Yu Y. On the shoulders of giants: a new dataset for pull-based development research. In: Proceedings of the 17th International Conference on Mining Software Repositories (MSR), 2020. 543–547

45 Jiang Y, Adams B, German D M. Will my patch make it? And how fast? Case study on the Linux kernel. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), 2013. 101–110

46 Hotti V, Koponen T. Defects in open source software maintenance-two case studies: Apache and Mozilla. In: Proceedings of Software Engineering Research and Practice, 2005. 688–693