Noname manuscript No.
(will be inserted by the editor)

FENSE: A Feature-Based Ensemble Modeling Approach to Cross-Project Just-in-Time Defect Prediction

Tanghaoran Zhang · Yue Yu ✉ · Xinjun Mao ✉ · Yao Lu · Zhixing Li · Huaimin Wang

Received: date / Accepted: date

Abstract Context: Just-in-time defect prediction (JITDP) leverages modern machine learning models to predict the defect-proneness of commits. Such models require adequate training data, which is unavailable in projects with short histories. To address this problem, cross-project methods reuse the data or models in other projects to make predictions, grounded on the assumption that they share similar defect-related features. However, these features are overlooked, which leads to unsatisfying model performance. Objective: This study aims to investigate the relationship between cross-project JITDP performance and project features, thereby improving the performance of cross-project models. Method: We propose a **F**eature-based **ENSE**mble modeling approach (FENSE) to cross-project JITDP. For a target project, FENSE pairs it to each source project and obtains 20 features. Leveraging them, it can predict the transferability of each off-the-shelf JITDP model. Then FENSE identifies the most transferable ones and combines them to make cross-project predictions. To achieve this, we conduct a large-scale empirical study of 113,906 project pairs in GitHub and investigate the impact of project features. Results: The results show that: (1) cross-project transferability is highly related to features including programming language and the defect ratio of the source project; (2) our feature-based model selection scheme can improve the cross-project JITDP performance by 10%; (3) FENSE outperforms other models on five evaluation measures without extra time and space costs. Conclusions: Our study suggests that project features can help identify powerful cross-project JITDP models and improve the performance of ensemble approaches.

T. Zhang · Y. Yu · X. Mao · Y. Lu · Z. Li · H. Wang
National University of Defense Technology, Changsha, Hunan, China
E-mail: {zhangthr, yuyue, xjmao, lizhixing15, hmwang}@nudt.edu.cn

✉ Yue Yu
National University of Defense Technology, Changsha, Hunan, China
E-mail: yuyue@nudt.edu.cn

✉ Xinjun Mao
National University of Defense Technology, Changsha, Hunan, China
E-mail: xjmao@nudt.edu.cn

Keywords Just-in-time defect prediction · Cross project · Ensemble learning · Model selection · Software project feature

1 Introduction

Software quality assurance is of vital importance in software development and operation. Due to limited testing resources and the popularity of open-source software (OSS), data-driven approaches are employed to aid traditional quality assurance activities. One of them is software defect prediction. It identifies defects in software through software metrics, such as lines of code, McCabe's cyclomatic complexity, etc., which has been widely adopted in industry and academia (Ostrand et al. 2005; Menzies et al. 2010; Lewis et al. 2013). However, it is usually exhausting to locate defects and find the developers who introduced them in large-scale software projects. Therefore, researchers put forward the concept *just-in-time defect prediction* (JITDP) (Mockus and Weiss 2000; Kim et al. 2008; Kamei et al. 2013), which is capable of predicting the risk of every code change right after its submission. It allows developers to review and fix the potential bugs in an efficient and resource-saving way, and hence it has received wide attention these years (Mockus and Weiss 2000; Aversano et al. 2007; Kamei et al. 2013; Yang et al. 2015; Kamei et al. 2016; Yang et al. 2016; Liu et al. 2017; Huang et al. 2017; Yan et al. 2020; Kondo et al. 2020; Wang et al. 2020).

However, training machine learning models requires adequate training data, otherwise they will not obtain satisfying performance. For JITDP tasks, few companies store their historical defect data for past projects. Even if enough data is provided, human inspection and annotation are still needed (Turhan et al. 2009). This process can be tedious and time-consuming, which imposes a severe challenge to JITDP. Inspired by transfer learning (Pan and Yang 2010), *cross-project defect prediction* (CPDP) approaches utilize data or models from other projects to make predictions. Based on reuse granularity, existing techniques can be divided into data-level and model-level transfer. The former manipulates data in source projects to build a training set similar to the target (Turhan et al. 2009; Ma et al. 2012; Nam et al. 2013; Kawata et al. 2015). However, due to limited data in target projects, data-level transfer suffers from over-fitting and needs retraining whenever new data arrives. The resulting costs of time and resources make this type of method not suitable for software with tight iterations. As for building a giant model by merging all cross-project data, its training cost is also exhausting. Additionally, in software companies, defect data is protected as a proprietary asset and thus inaccessible for developers.

Model-level transfer is proposed to alleviate this problem. It selects or combines off-the-shelf models trained on other projects to predict risky commits in the target project. Model-level transfer is simple and applicable, but its performance is discouraging. A common solution is to apply ensemble modeling (Fukushima et al. 2014; Kamei et al. 2016; Catolino et al. 2019; Tabassum et al. 2020). However, ensemble models behave differently and sometimes even perform worse than traditional ones. The reason lies in the selection of base learners and ensemble techniques. In addition, Kamei et al.'s approach (2016) based on project similarity rarely improves the performance, which suggests that the relationship between model performance and project similarity is not straightforward. Thus, a thorough investigation of factors in cross-project performance is required.

Therefore, we propose a feature-based ensemble modeling approach, FENSE, to perform the cross-project JITDP task in this paper. Given a project set and a target project, FENSE first pairs the target with each source project on which a JITDP model has already been trained. Then it obtains 20 features in four levels as inputs and predicts the transferability of each off-the-shelf JITDP model. Transferability refers to the estimated performance when a model is applied to the target. FENSE identifies the most transferable JITDP models and generates a sequence taking advantage of their features. Finally, the models in sequence are integrated to predict the risk of commits in the target project.

To investigate the factors in cross-project transferability, we conduct a large-scale empirical study on OSS projects in GitHub. We first collect the development history of 831 projects in GitHub and employ an improved SZZ algorithm to annotate each commit. After preprocessing, we construct 113,906 transferring pairs of 338 projects. For each pair, 20 features are extracted. Their impacts on cross-project JITDP performance are analyzed by regression analysis. Finally, we validate our approach on a test set with 67 projects and compare it to other cross-project JITDP methods to prove its effectiveness.

To be specific, we address the following four research questions:

(RQ1) What are the factors in the cross-project transferability of JITDP models?

(RQ2) Is our feature-based model selection scheme effective? How much benefit can it bring to the cross-project performance of JITDP models?

(RQ3) Can ensemble modeling improve the cross-project performance of a single JITDP model? What are the effects of different combination methods and integration scales?

(RQ4) How does FENSE perform compared to other cross-project JITDP models?

These four RQs test the validity of FENSE in sequential order. In RQ1, we investigate our proposed features to measure their impacts on cross-project transferability. The programming language and the defect ratio of the source project are two influential factors in transferability. Moreover, the conditional R^2 of our regression analysis is 0.844, which shows our features are expressive. In RQ2, we find out that our feature-based model selection scheme prioritizes highly transferable models to demonstrate its effectiveness. In RQ3, we discuss the effect of each component in the ensemble modeling of FENSE. Finally, in RQ4, we evaluate the overall performance of FENSE by comparing it to other five cross-project JITDP approaches. The results show that FENSE outperforms other models on 5 out of 6 evaluation measures, particularly on recall and effort-aware metrics. Its time and space costs are also acceptable.

The overall framework of our study is shown in Figure 1 and illustrated in Section 3.

The contributions of this paper include:

– We collect 8,169,560 commits from 831 projects in GitHub and annotate them using an improved SZZ algorithm, which composes a large-scale cross-project

JITDP dataset. Our study proves that model-level transfer actually performs well when the dataset is large enough.
- We extract 20 project features for each project pair and build a regression model to investigate their impacts on cross-project transferability, which is overlooked by previous studies. The programming language and the defect ratio of the source project are two influential factors in transferability.
- We propose a feature-based ensemble modeling approach, FENSE, to perform cross-project JITDP tasks. Its model selection scheme leverages the contextual knowledge of projects to integrate multiple trained models with higher transferability. The results prove that FENSE can prioritize learners with higher transferability for different targets and perform best on five of six evaluation metrics.
- We find that the effect of ensemble modeling is not comparable to the model selection scheme. Ensemble models can achieve their expected performance by integrating several models. Moreover, their effort-aware metrics and cost-effectiveness decrease as the integration scale grows. Thus, merely increasing integration scale without proper consideration of software features cannot benefit cross-project JITDP.

The rest of this paper is organized as follows: Section 2 discusses the background and related work of CPDP and JITDP. Section 3 presents our methodology of seeking factors in cross-project transferability and approaches to building the cross-project model. Section 4 describes our experimental design, including dataset construction, model building, evaluation techniques, and comparing methods. Results are illustrated in Section 5. Section 6 summarizes the threats to validity of our methods. Finally, Section 7 concludes our work.

2 Related Work

2.1 Just-in-Time Defect Prediction

Just-in-time defect prediction, known as a change-level quality assurance method, uses change metrics, e.g., the number of modified files, rather than module-level metrics, e.g., the number of methods used in a given class or module, to predict the defect-proneness of commits. It is regarded as a resource-saving way to assist software quality assurance activities in practice.

JITDP can be traced back to Mockus and Weiss's work (2000). They hold that changes are the most fundamental and immediate concern in a software project. Kim et al. (2008) regarded predicting bugs in software changes as a classification task and introduced a change-level defect prediction technique. The concept 'just-in-time quality assurance' was proposed by Kamei et al. (2013). They conducted an empirical study on six open-source and five commercial projects from an effort-aware view. Several change measures in five aspects (diffusion, size, purpose, history, and experience) were summarized to predict the risk of changes. They have been widely applied in recent studies (Yang et al. 2015; Kamei et al. 2016; Yang et al. 2016; Liu et al. 2017; Yang et al. 2017; Huang et al. 2017; Yan et al. 2020; Kondo et al. 2020).

Novel techniques are constantly being adopted to JITDP by researchers, i.e., ensemble learning and deep learning approaches are used to generate features for defect prediction. Yang et al. (2017) proposed a two-layer ensemble learning approach (TLEL) to improve the performance of JITDP. The authors combined the bagging and stacking strategies to build a two-layer model, which proved effective and robust. Chen et al. (2018) considered JITDP as a multi-objective optimization problem and proposed an unsupervised approach, MULTI, to identify more buggy changes with less effort. Their results suggested that MULTI performs significantly better than all the state-of-the-art methods. Hoang et al. (2019) proposed an end-to-end deep learning framework for JITDP. Their model used a convolutional neural network to extract features from commit messages and code changes, outperforming state-of-the-art methods. Wang et al. (2020) considered using a deep belief network to extract the semantics of code to perform defect prediction. They validated their results on two file-level and two change-level prediction tasks, which improves current results significantly. Li et al. (2020) proposed a semi-supervised model named EATT for effort-aware JITDP. It leveraged the idea of tri-training to exploit more unlabeled defect data. Compared with the state-of-the-art supervised and unsupervised models, EATT has a considerable advantage over them.

Hoang et al. (2020) proposed a neural model, CC2Vec, to represent the semantics of code changes. They evaluated the results produced by CC2Vec in JITDP tasks, which outperforms DeepJIT. Several researchers have followed their work and provided new approaches to JITDP recently. Pornprasit and Tantithamthavorn (2021) pointed out that CC2Vec requires all unlabelled testing data beforehand and hence does not follow the principles of JITDP. Their replication study demonstrated that the performance of CC2Vec decreases heavily when excluding the testing set. To this end, they proposed JITLine, a JITDP approach that can also identify defective lines. It is more fine-grained and more accurate than other baseline approaches. Zeng et al. (2021) validated CC2Vec on a large-scale dataset. Their results showed that a traditional JITDP model that only considers the added-line-number feature can outperform CC2Vec and be a hundred times faster.

Previous studies imply that JITDP is a practical way to ensure software quality with finer granularity and high efficiency. However, these models require a large training corpus of within-project data to perform better. As for projects with few historical data, local JITDP methods are often not available. Therefore, this paper studies JITDP models in a cross-project context.

2.2 Cross-Project Defect Prediction

Cross-project defect prediction is a popular sub-field concerned by researchers. The lack of training data makes it impossible to build machine learning models. CPDP approaches are proposed to solve this problem by reusing available data or models from other projects to make predictions. This idea was first introduced in Briand et al.'s study (2002), in which they considered reusing the fault-detecting models across software systems. Their results suggested that the applicability of cross-project models is far from straightforward because of system differences.

Zimmermann et al. (2009) ran cross-project predictions for 12 real-world applications and analyzed the effects of 40 software metrics on cross-project predictability. Among 622 cross-project cases, they only got 21 satisfying results, suggesting simple using models from related projects cannot lead to better model performance. As for software metrics, they also wondered how to find the right set of metrics to make cross-project predictions.

Their dedicated work enlightened us to further study the factors in cross-project transferability, particularly in a large-scale dataset. With the big data from OSS projects, we can model the influence of different project features more accurately through regression analysis.

To improve the cross-project performance, prior studies proposed two types of CPDP approaches leveraging the idea of transfer learning. One is data-level transfer. It refers to leveraging data from other projects to build a training set for targets. Turhan et al. (2009) applied nearest neighbor filtering to cross-project data and used a similar part for defect prediction in target projects. Kawata et al. (2015) argued that the predicting performance of CPDP relies on the quality of subset selection heavily. Hence, they proposed a new relevancy filter based on DBSCAN, an advanced clustering algorithm. Ma et al. (2012) proposed an algorithm called Transfer Naive Bayes (TNB) to construct a training set by weighing cross-project data. Nam et al. (2013) found that the difference in feature distributions between two projects led to poor CPDP performance. They employed Transfer Component Analysis (TCA) and proposed TCA+ to find a latent feature space where data distributions of source and target projects were similar. These approaches tried to build a training set similar to the target from data distribution and achieved good results. However, the defect data is often treated as a proprietary asset, thereby inaccessible for developers. Besides, data-level transfer suffers from conclusion instability (Krishna et al. 2016), which refers to frequently retraining as long as new data comes. The resulting costs of time and resources make this type of method not suitable for software with tight iterations.

Another data-level method called data-merging directly combines all cross-project data as a training set (Kamei et al. 2016; Tabassum 2020). Different from the aforementioned similarity-based transfer methods (Turhan et al. 2009; Kawata et al. 2015), it simply assumes that the generalization ability of a model can be achieved when defect data is adequate, thereby achieving good results on the target. Kamei et al. (2016) suggested that such a simple approach can yield models performing well in a cross-project context. Tabassum et al. (2020) investigated the effects of cross-project data in realistic online JITDP scenarios through data merging. They concluded that using cross-project data can benefit JITDP at the initial phase and even when much within-project data is available. Similar to the methods above, data-merging also requires a plethora of training data from different projects, which is often unavailable. At the same time, it is time-consuming to train such a large model.

Another type of CPDP approach is model-level transfer. It selects or combines defect prediction models that have been trained on other projects to predict the risk of commits in the target project. It is more straightforward and applicable than data-level transfer and enables us to make predictions without necessary access to defect data. However, the limitation of model-level transfer is its low performance. A common solution is to leverage ensemble modeling. Fukushima et al. (2014) empirically evaluated cross-project JITDP models on 11 OSS projects and found that the ensemble model has the best performance. Catolino et al. (2019) conducted an empirical study of 14 mobile applications and 42,543 commits, where four traditional learners and four ensemble models were evaluated. The results manifested that Naive-Bayes performs best in these models rather than ensemble ones. Tabassum et al. (2020) held that the ensemble approach is detrimental to the performance in an online scenario.

The results above suggest that researchers have mixed attitudes to the effectiveness of ensemble models. The reason lies in the selection of base learners or ensemble techniques. Kamei et al. (2016) considered selecting several models that shared higher project similarity with the target project and using similarity-weighted voting for the ensemble model. However, its performance was not significantly improved, suggesting that the relationship between model performance and project similarity is not straightforward. A thorough investigation of factors in cross-project performance is required.

Apart from the ensemble modeling, Krishna et al. (2016) proposed a single-model transfer approach called 'Bellwether' to simplify the CPDP task. Bellwether is the model that produces the best result among a project community. Their results showed that bellwether exists and generates more accurate predictions than local models. It demonstrated that there are models which can perform well in a cross-project context, but the statistics cannot predict them. We believe a large-scale dataset and the consideration of project features can benefit the identification of good cross-project models.

Therefore, a powerful model selection scheme should be proposed. It should investigate the factors in model performance among a large scale of projects and various project features, which is our focus in this paper.

3 Methodology

Our research methodology follows a quantitative approach. The overall framework is shown in Figure 1. In general, three main activities are included in our research: (1) building a large-scale JITDP dataset from OSS projects in GitHub; (2) conducting a regression analysis to investigate the impact of project features on cross-project transferability; (3) building an ensemble model to perform the cross-project JITDP task. Specifically, we clone hundreds of GitHub repositories and crawl relative project data, e.g., issues and contributors. We carefully clean our repositories by several filtration rules. To obtain the label of each historical commit, we annotate the data with an improved SZZ algorithm. The process of dataset construction will be explicated in Section 4.1. We extract project features in four different levels for all projects and train their within-project JITDP models with labeled data. To figure out the factors in cross-project transferability, we use a linear mixed-effect model for regression analysis. Unlike existing ensemble methods, we leverage this contextual knowledge for model selection by prioritizing top-K trained models with higher transferability. A model sequence is generated as the input for the ensemble model. Finally, we integrate them with designed combination methods such as voting.

3.1 Regression Analysis of Cross-Project Transferability

As we introduced in Section 2.2, model-level transfer approaches are preferred when defect data is unavailable. Prior studies proposed several project features to characterize the software projects. The most common and popular ones include

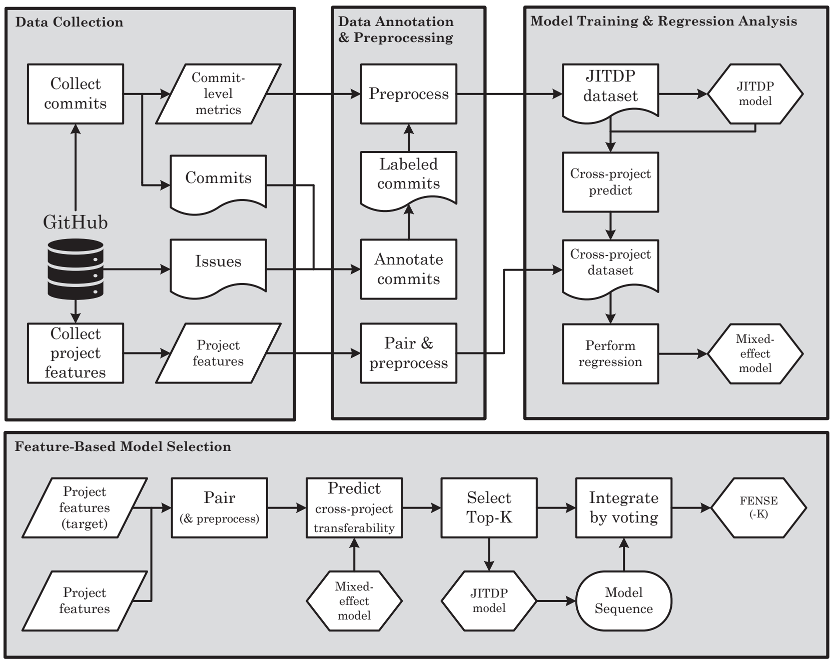


Fig. 1 Our proposed framework for building a JITDP dataset and applying FENSE to the cross-project JITDP task

the application domain, programming language, the total number of lines of code, and the number of developers, which have been investigated in both proprietary and OSS projects (Capiluppi et al. 2003; Zimmermann et al. 2009; Zhang et al. 2013; Zhang et al. 2014; Kamei et al.2016; Lin et al. 2021).

Based on their efforts, our study adopts or refines their proposed metrics and considers more features for OSS projects in GitHub to characterize them as comprehensively as possible. Importantly, we study their effects on cross-project JITDP on a large-scale dataset to retrieve more valuable and convincing results.

To achieve this, we pair each two projects, extract project features from four levels, and conduct a regression analysis to figure out the factors in cross-project JITDP performance. In our expectation, the resulting knowledge should enable us to identify models with higher cross-project transferability. Our ensemble modeling approach based on this will be discussed in Section 3.2.

Given a project set S, we pair every two projects as (P_i, P_j). The former (P_i) is called the source project, whose JITDP model M_i is then used for predicting the risk of commits in the target project P_j. Its performance y_{ij} (measured by ROC-AUC score in our study) shows the cross-project transferability of M_i. For the project pair (P_i, P_j), we propose several features as predictors, including numerical and categorical ones, to quantify the features which may have impacts on the response variable y_{ij}.

To perform linear regression, we should guarantee that our data meet several assumptions (Farrar and Glauber 1967), including the normality of population, the independence of factors, the homoscedasticity of residuals, and the independence between observations. The numerical variables are first log-transformed in our study to stabilize variance and reduce heteroscedasticity (Cohen et al. 2014). To ascertain the independence of predictors, we evaluate the variance inflation factors (VIFs) and the Spearman correlation coefficients ρ. Our paper considers that VIFs lower than 3 and ρ lower than 0.8 indicate the absence of multicollinearity, which was suggested by Cohen et al. (2012) and prior studies (Bettenburg et al. 2010; Kamei et al. 2016; Jiarpakdee et al. 2018).

Given that our data is not independent (a model is used to predict the likelihood of defect-prone commits in multiple target projects, and a project is predicted by multiple models), we group them and build a linear mixed-effect model. In particular, we use the random intercept model to consider the effect of different projects. Unique identifiers (i.e., the project indexes) are regarded as random intercepts to eliminate the correlation between different observations, while predictors are regarded as fixed effects. Here we use the implementation of `lmer` function in the `lme4` package of R.

To describe software projects comprehensively, we extract several features to measure the relationship between a project pair. Here some features depend on both source and target (e.g., same_owner_type) while others are independent. They are classified into four levels: model level, project level, social level, and technical level. Model-level metrics reflect the features of the JITDP model M_i, while the other three levels portray the similarity between the pair (P_i, P_j). We believe that how a model is trained and project contextual knowledge are related to cross-project transferability.

A) Model-level features.

n_commits_src: This metric refers to the number of commits in the training set of M_i. In general, more training samples provide a more accurate representation of true data distribution, leading to a better generalization ability.

model_performance_src: This metric refers to the performance of M_i. We believe a good within-project model can also provide accurate predictions for other projects. Although Kamei et al. (2016) found that the performance of local JITDP models is not a strong indicator of cross-project JITDP, we intend to validate their conclusion on a large-scale dataset in our paper.

defect_ratio_src: This metric refers to the ratio of positive samples, that is, the proportion of buggy commits in the source project P_i. It indicates the degree of class imbalance. The higher one means M_i is trained using a more balanced set, thus it may have a positive influence on its cross-project performance.

Note that these three features are only related to the source project P_i and its model M_i, hence users do not need to know those of target projects in practice.

B) Project-level features.

project_popularity: This metric refers to the popularity of P_i (or P_j) (Capiluppi et al. 2003). This paper uses the number of 'Watch' of a GitHub repository to reflect its popularity.

project_age: This metric refers to the period from the creation of P_i (or P_j) to the time we collected it in days. We use this metric to reflect its maturity, which is also used by Capiluppu et al. (2003) and Zhang et al. (2014).

same_owner_type: This metric refers to whether the owner types of P_i and P_j are the same. The owner type can be 'User' or 'Organization' in GitHub. Projects with the owner type of 'User' are personal repositories, while 'Organization' projects allow developers to collaborate across many projects and customize settings.

same_license: This metric refers to whether the open-source licenses are the same for P_i and P_j. A license shows what people can or cannot do with a project. Common open-source licenses include Apache license 2.0, GPL, MIT, etc.

same_language: This metric refers to whether the programming languages of P_i and P_j are the same. This feature is commonly used by prior studies (Capiluppi et al. 2003; Zimmermann et al. 2009; Zhang et al. 2013; Zhang et al. 2014; Kamei et al.2016; Lin et al. 2021). We suspect cross-language predictions are different from within-language predictions. This paper collects OSS projects in Java, C/C++, Python. C and C++ are regarded as the same language. We believe M_i has higher cross-project transferability if P_i and P_j are programmed in the same language.

textual_similarity: This metric refers to the textual similarity of project description and readme files between P_i and P_j. Developers describe the application domain and functions of a software project in these two parts, which is concerned by prior researchers (Capiluppi et al. 2003; Zimmermann et al. 2009; Zhang et al. 2013; Zhang et al. 2014; Kamei et al.2016; Lin et al. 2021). We believe the textual similarity between P_i and P_j highly summarizes the domain and functional information between projects. Thus a higher value of it indicates that M_i may perform better when predicting P_j. In this paper, we first concatenate the brief description and the readme text. Then we clean the text by removing the tags, symbols with no meaning, and stopwords in English. We use Porter Stemmer in NLTK[1] to normalize the terms. Term frequency-inverse document frequency (TF-IDF) is used to vectorize the text. Finally, the textual similarity is measured by the cosine similarity between two vectors.

C) Social-level features.

For social-level features, prior studies (Capiluppi et al. 2003; Zhang et al. 2014) use the number of developers to measure the size of a software system, which is also used by Zimmermann et al. (2009), Kamei et al. (2016), and Lin et al. (2021). Our paper refines them using information extracted from the open-source community. We use the number of core members and external contributors instead of 'developers' to distinguish their roles in the OSS development process. At the same time, we believe that human behaviors are more related to the cross-project JITDP rather than only the size of a project.

n_core: This metric refers to the number of core members in P_i (or P_j). Core members are people with higher permission levels, e.g., they can close an issue or merge a pull request. In GitHub, core members are the owner and the collaborators of a project.

n_external: This metric refers to the number of external contributors in P_i (or P_j). They can make contributions by submitting pull requests, reporting issues, giving comments, etc.

core_diff: This metric refers to the difference in the number of core members between P_i and P_j.

external_diff: This metric refers to the difference in the number of external contributors between P_i and P_j.

contributor_intersection: This metric refers to the number of co-exist contributors (include core members and external contributors) in P_i and P_j. If the same

[1] https://www.nltk.org/

group of developers contribute to both projects, they will likely generate similar development data. M_i may have better cross-project performance when their data distributions are similar.

entropy_diff: This metric refers to the difference of P_i's and P_j's contribution entropy values. Contribution entropy represents the distribution of the number of activities among all contributors. It can be calculated by:

$$entropy = -\sum_{k=1}^{n} c_k \times \log_n c_k \tag{1}$$

$$c_k = \frac{contributions_k}{\sum_{l=1}^{n} contributions_l} \tag{2}$$

where n refers to the number of contributors, and c_k refers to the percentage of the k-th contributor's contributions. The value of *entropy* closer to 1 implies that the contributions from each contributor are almost the same, while 0 means that contributions are almost from a single person.

D) Technical-level features.

code_size: This metric refers to the code size of P_i (or P_j) in kilobyte, which is also suggested by Capiluppi et al. (2003), Zimmermann et al. (2009), Zhang et al. (2013), Zhang et al. (2014), Kamei et al. (2016), and Lin et al. (2021).

code_size_diff: This metric refers to the difference in code size between P_i and P_j.

n_dependencies: This metric refers to the number of project dependencies in P_i (or P_j). Dependencies are the relationships between packages. In this paper, we extract the dependencies of each GitHub repository through libraries.io[2].

dependency_intersection: This metric refers to the number of co-exist project dependencies in P_i and P_j.

dependency_diff: This metric refers to the difference in the number of code dependencies of P_i and P_j.

Our paper directly adopts several metrics from prior work (Capiluppi et al. 2003; Zimmermann et al. 2009; Zhang et al. 2013; Zhang et al. 2014; Kamei et al. 2016; Lin et al. 2021), i.e., the programming language, the code size, and the project age, because they have intuitive relations with cross-project transferability. For measures that are not available or difficult to obtain in OSS projects, i.e., domain, company, and intended audience, we use textual similarity in the project description and readme files to consider them as a whole.

Besides, we supplement some features during data collection by ourselves. In RQ1, we discuss each feature's effect on cross-project transferability, so we consider adding more features to characterize the projects as comprehensively as possible, such as the project popularity, the same owner type, the same license, and project dependencies. Although the relationship between the cross-project transferability and these features is not straightforward, they can describe the characteristics of OSS projects to some extent, for instance, their organizational structures. We hope they can help improve the prediction of cross-project transferability. These features are all accessible from GitHub.

[2] http://libraries.io/

For numerical features, i.e., the number of developers and the code size, previous studies separate them into levels or groups (least/less/more/most) to characterize the similarity between projects, while we calculate their precise values concerning their differences or intersection and apply them to our regression analysis, e.g., core_diff.

3.2 Cross-Project JITDP Based on Ensemble Modeling

Prior studies have mixed attitudes to the effectiveness of ensemble models on cross-project JITDP. The reason lies in the selection of base learners or ensemble techniques. This paper reconsiders the cross-project problem and proposes a feature-based ensemble modeling approach named FENSE. The key idea of FENSE is to seek and integrate the 'best' models with higher cross-project performance. Leveraging the knowledge of project features, we first estimate the cross-project performance of trained models. For different targets, we select the top-K and generate a model sequence. This process is called feature-based model selection in our study. Then, we integrate the models using different combination methods when the integration scale (denoted by K) changes to predict the defect-proneness of commits. Since the selected models can be directly employed in the prediction process, the model application process is also computationally efficient.

Given a training project set $S_{train}(P_i \in S_{train})$, a test project set $S_{test}(P_T \in S_{test})$ and a regression model LME fitted on S_{train}, the model selection and application process are described as follows (can also refer to Figure 1):

1. **Pair:** For each source project P_i in S_{train}, pair it to the target project P_T and then extract the features of the pair (P_i, P_T). A preprocessing is adopted to meet the requirement of LME.
2. **Predict:** Use LME to predict M_i's cross-project transferability on P_T.
3. **Select:** Prioritize top-K models with higher estimated cross-project performance. Here a model sequence is generated for next step.
4. **Integrate:** Use selected models to predict the risk of commits for each P_T and combine them by voting. Different combination methods and integration scales (K) are discussed.

Furthermore, the simplicity of FENSE needs to be highlighted. For a target project, we only need to obtain its features and integrate predicting results of several trained models, while no retraining is required. Note that we can also update our local models and linear mixed-effect model to include more up-to-date data as existing projects evolve, which can be done in a distributed manner.

4 Experimental Design

4.1 Dataset

To investigate the factors in cross-project transferabililty, we need a plethora of projects with adequate historical data. It is unrealistic to obtain such a large amount of data from the industry. Thanks to the prosperity of OSS projects on online code hosting platforms, e.g., GitHub, GitLab, BitBucket, etc., we are exposed to a series of energetic software with rich development data. Leveraging the open-source community, we build a large-scale JITDP dataset by carefully selecting projects and annotating their commits. In this work, we demonstrate the performance of our approach on this crafted dataset. Developers can directly use their off-the-shelf prediction models or train models by themselves in real scenarios.

4.1.1 Data Collection

We build a JITDP dataset comprised of 900 OSS projects in GitHub. They are written in three different programming languages (Java, C/C++, Python, 300 respectively), which guarantees that they are code hosting projects. Hence, we can perform defect prediction tasks on the historical changes. Moreover, to ensure the quality of annotation, we use four filtering rules as follows:

1. Projects are not forks of existing repositories. This rule guarantees that there are no identical repositories in our dataset. Repetitive data may perturb our results.
2. Projects have at least 500 issues. This rule guarantees that we have enough issue data to annotate commit samples.
3. Projects are ranked and collected by stars. This rule guarantees the popularity and maturity of projects. These projects are more likely to keep high-quality development histories, which benefits our data annotation and analysis process. However, some popular projects with attractive features are not notable (e.g., cloud-to-butt). We winnow them out naturally after preprocessing, as they often lack enough historical data.
4. Projects are described in English. This rule guarantees the validity of textual information extracted from those projects because several natural language processing methods are used when collecting software metrics.

After filtration, we obtain 256 Java projects, 291 C/C++ projects, and 284 Python projects.

4.1.2 Data Annotation

We collect and annotate all the historical commits of studied projects from birth to March 16, 2021. The most common way to label the commits is using the SZZ algorithm (Sliwerski et al. 2005). However, the risks of SZZ have been concerned by previous studies for ages. To avoid the common pitfalls, we carefully employ SZZ by double-checking after each step, and its process is described in the following.

Identifying bug reports. The first step in the SZZ algorithm is to find bug reports through the issue tracking system. GitHub ITS[3] uses labels to mark different issue types, such as enhancement, feature, etc. However, most of them are user-defined, and their naming rules differ completely in different projects. Hence, we first collect all the issue labels in our dataset and manually inspect 300 of them with the highest frequency. Keywords such as 'bug' or 'defect' are included in those buggy labels.

We use regular expressions to distinguish bug reports from other not bug-related issues. Note that labels such as 'not a bug' need to be carefully winnowed out.

Identifying bug-fixing commits. Bug-fixing commits (or bug fixes) are changes that repair the reported bugs during the production and maintenance phases. Common practices show that a bug fix often contains a bug report number and bug-related keywords in the commit message (Sliwerski et al. 2005). GitHub also supports users to close an issue by entering a commit message like 'Fixes #47'[4]. Therefore, we identify commits with keywords or links to bug reports as bug fixes. Given that developers may forget to record the bug report numbers, we adopt the idea of ReLink (Wu et al. 2011) to associate potential bug fixes with bug reports heuristically.

However, the operations above may lead to the overestimation of bug fixes. To guarantee the reliability of bug fix identification, we adopt two filtration schemes. For the one, non-functional revisions and revisions made in non-source files (i.e., documentation, test codes) cannot fix any bugs. Non-functional revisions include changes in blank lines or comments and inline format changes. Thus, we treat these changes as non-fixes.

For the other, commits with a large number of file changes (NF) need to be reconsidered. Kim et al. (2006) consider it suspicious for the reason that not all changes in a large-scale revision are related to the bug-fixing activities, as large commits are often perfective than corrective (Hindle et al. 2008). Moreover, such commits are likely to be tangled changes (Herzig et al. 2013) when a list of tasks are performed in one commit. In our paper, we identify outliers in NF and regard their corresponding bug-introducing commits traced by SZZ as suspects, as we cannot tell whether they result in bug fixes or not. These suspects are removed from our dataset.

In our dataset, we regard commits whose NF are above the Upper Median Absolute Deviation (Upper MAD) as the outliers of each project. MAD is robust when analyzing non-normal data, and it is also a good indicator to identify outliers (Leys et al. 2013).

Identifying bug-introducing commits. As mentioned before, non-functional revisions and revisions made in non-source files cannot fix any bugs. So we neglect those lines and files in SZZ. Apart from this, we also filter out three kinds of suspects in this step. As for version-control-related commits, such as merge-changes, they only merge the changes from another branch and hence cannot induce fixes. We add these changes to suspects. Additionally, if a change was committed after its associated bug reports were made, it should not be the cause of the reported bugs (Kim et al. 2006). We also regard them as suspects. In da Costa et al.'s study (2017), they held that the future impact of a commit is a critical standard for the SZZ algorithm. For a bug-introducing commit, it is improbable to introduce too many bugs, and the time span of future bugs should not be too long. Therefore, we collect the number of bug reports related to each bug-introducing commit and calculate the time span. Subsequently, we use Upper MAD to identify outliers and add them to suspects.

[3] https://guides.GitHub.com/features/issues/

[4] https://GitHub.blog/2013-01-22-closing-issues-via-commit-messages/

Table 1 An overview of our dataset

		#Commits	#Buggy	%Buggy
Java	Mean	6491.1	1044.97	16.1%
	Minimum	390	147	5.59%
	Median	4992	665	14.85%
	Maximum	29241	5719	62.81%
C/C++	Mean	8646.55	1426.48	16.5%
	Minimum	558	143	4.39%
	Median	4663	787	17.14%
	Maximum	58562	12726	67.42%
Python	Mean	5438.27	975.47	17.94%
	Minimum	497	164	6.55%
	Median	3004	530	18.11%
	Maximum	46205	10515	54.81%

In this step, we use GitPython and PyDriller (Spadini et al. 2018) as auxiliary tools to mine git repositories. Due to the verification latency (Cabral et al. 2019, Tabassum et al. 2020), the latest commits collected may not be correctly labeled. By exploring into our data, over half of bugs were reported and fixed in 90 days, which refers to the defect discovery delay in (Cabral et al. 2019). To ensure that our data have credible labels, we only use the commits before December 16, 2020.

4.2 Within-Project Model Building

Before our analysis, we build local JITDP models for all projects for two reasons. One is that they are considered ready-made models for cross-project JITDP and base learners of our ensemble approach. For the other, local model performance is a strong indicator of cross-project transferability in our regression analysis.

As a matter of fact, this step is unnecessary when applying our approach if developers have already had some elaborate models. In this paper, to demonstrate our idea on ensemble modeling, we explicate how we build JITDP models from scratch. However, these within-project models are not available when predicting projects with little historical data in real scenarios.

4.2.1 Preprocessing

We split the commits of each project into training and test set with a ratio of 4 to 1. Before training JITDP models, we employ two preprocessing techniques. The first is to handle the class imbalance. It is believed that JITDP suffers heavily from the data imbalance problem (Tan et al. 2015). As is shown in Table 1, defective changes are also far less than clean ones in our dataset, which only account for 16.1%, 16.5%, 17.94% in Java, C/C++, and Python projects, respectively. If no proper rebalancing techniques are used, a learner that predicts all the samples as negative could reach high accuracy. Therefore, we perform an under-sampling operation in the training set by deleting instances randomly such that the ratio of the minority class is between 20% and 80%. Similar under-sampling methods are also used in Yang et al. (2015), Kamei et al. (2016), and Kondo et al. (2020) 's work.

Table 2 A brief summary of commit-level metrics in prior work

Aspect	Metric	Description
Diffusion	NS	Number of subsystems modified (Mockus and Weiss 2000)
	ND	Number of modules modified (Mockus and Weiss 2000)
	NF	Number of files modified (Nagappan et al. 2006)
	Entropy	Distribution of modified code across each file (Hassan 2009)
Size	LA	Lines of code added (Nagappan and Ball 2005)
	LD	Lines of code deleted (Nagappan and Ball 2005)
	LT	Total lines of code in a file (Koru et al. 2009)
Purpose	FIX	Whether or not current change is a bug fix (Guo et al. 2010; Purushothaman and Perry 2005)
History	NDEV	Number of developers that changed the modified files (Matsumoto et al. 2010)
	AGE	Average interval between last and current change (Graves et al. 2000)
	NUC	Number of unique changes of the modified files (Hassan 2009)
Experience	EXP	Developers experience (Mockus and Weiss 2000)
	REXP	Recent experience
	SEXP	Developer experience on a subsystem

For the second, we employ filtration to the training and test samples. We filter out projects with less than 500 training commits to guarantee that each model has sufficient training data. At the same time, we filter out projects with less than 30 positive (defect-prone) test samples to mitigate the sensitivity of evaluation metrics (i.e., recall). Here we make a trade-off between the size of our project set and the threshold mentioned above to guarantee the validity of our regression analysis. Table 1 shows the statistics of our dataset.

4.2.2 Commit-level Metrics Extraction

Previous research by Kamei et al. (2013) proposed 14 commit-level metrics of 5 dimensions, including NS, ND, NF and Entropy in Diffusion aspect, LA, LD, and LT in Size aspect, FIX in Purpose aspect, NDEV, AGE, and NUC in History aspect, and EXP, REXP, SEXP in Experience aspect. Table 2 shows their brief descriptions. These metrics are widely applied in recent studies (Yang et al. 2015; Kamei et al. 2016; Yang et al. 2016; Liu et al. 2017; Yang et al. 2017; Huang et al. 2017; Yan et al. 2020; Kondo et al. 2020).

This paper excludes several metrics and adopts new ones due to the cross-project context and data characteristics.

Firstly, we remove metrics in the History and Experience aspects because they are not accessible from software projects without any change histories (Fukushima et al. 2014; Kamei et al. 2016). Additionally, newly developed projects are common targets for cross-project JITDP. Without the historical data, predictions cannot be performed for models that consider History and Experience metrics.

Secondly, we remove NS and ND from the Diffusion aspect. For NS, we manually check the structure of projects in our dataset and find that nearly half of them are not organized into subsystems. For the others, subsystems can be placed in root directories, under the 'src' or 'modules' paths, or even customized directories. Here we exclude these metrics to avoid using data unfairly. As ND is highly correlated with NF, we keep a more fundamental metric NF.

Thirdly, although multicollinearity still exists for some metrics (e.g., LA and LD), we give up using additional techniques, e.g., Relative Churn (Nagappan and Ball 2005) for the following reasons. We use Random Forest as our JITDP model, which is less sensitive to multicollinearity because of its randomized sampling of features. Moreover, we prefer keeping the meaningful metrics rather than confusing them with others, e.g., lines of code added (LA) is the root cause of bug introduction in the SZZ algorithm.

Finally, Shihab et al. (2012) find that the number of bug reports linked to a change (NBR) is also a good predictor for the risk of change. Similar to FIX, it shows the bug fixing purpose of a software commit, so we adopt it as a supplementary metric in the Purpose aspect.

To sum up, we use NF, Entropy, LA, LD, LT, FIX, and NBR as our predictors for defective changes.

4.2.3 Model Training

We choose Random Forest (Ho 1995) as our learning model for within-project prediction. Random Forest is an ensemble model built by several decision trees. Nodes in these trees consist of a random subset of all features, so this model is less sensitive to correlations. Random Forest is reported to be more accurate and robust than other models for defect prediction (Kamei et al. 2016).

We consider tuning four parameters for each model: num_of_trees, max_features, min_sample_split, and max_samples. Since prior studies (Tosun and Bener 2009; Tantithamthavorn and Hassan 2018) pointed out that default parameters are often suboptimal for defective prediction tasks and recommended automated parameter optimization, we use grid search to find their best values. Scikit-learn (Pedregosa et al. 2011) is used in our implementation of FENSE and other comparing methods.

4.3 Evaluation

4.3.1 Evaluating Regression Model

Our study uses regression analysis to identify influential factors in cross-project transferability. Hence, we perform the ANOVA Type-II test to measure the effects of our proposed metrics. Each metric's χ^2 value represents its impact on the response variable. The larger the value of χ^2 is, the more influential the corresponding metric is. In addition, we also report the significance level of each metric. When the probability of an event is lower than a significance level, we can claim that its corresponding metric has a statistically significant impact on the response variable.

Apart from this, we calculate the conditional R^2 proposed by Nakagawa et al. (2013) to evaluate the goodness-of-fit of our linear mixed-effect model. It indicates the proportion of total variance explained by fixed and random effects. We use the implementation of the R function in `MuMIn` package.

4.3.2 Evaluating JITDP Models

Evaluation Settings JITDP is a time-sensitive process as commits are in time order. Bug generation, identification, and fix also happen sequentially. Previous studies reported that several cross-validation techniques tend to mix data in the past and future (Tan et al. 2015). However, future commits are unavailable when applying the prediction models. Hence the results of cross-validation are not convincing. In our study, we use Time Series Split, an approach that splits the training and test set of each project following the time order (see Figure 2), which was also employed in Yan et al. (2020) and Wang et al.'s work (2020). T_{birth} is the creation time of a project, while T_{test} is the time before which our data have credible labels as we mentioned in Section 4.1.2. Commits before time T_s are training samples, while commits after time T_s are test samples. We also adopt Time Series Split when fine-tuning the model for a similar purpose.

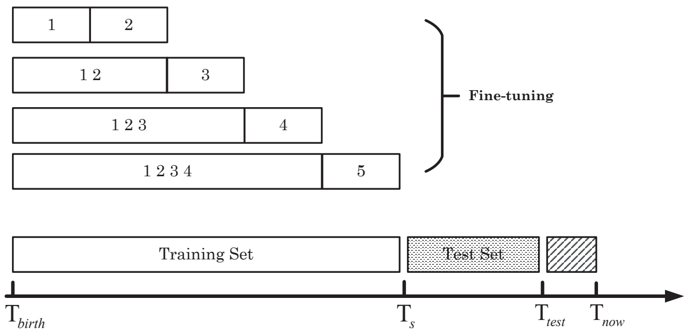


Fig. 2 The split of training and test set following the time order

Evaluation Metrics Traditional machine learning metrics are commonly used to measure the defect predicting performance in previous studies (Kim et al.2008; Kamei et al. 2013; Tan et al. 2015; Kamei et al. 2016; Wang et al. 2020; Tabassum et al. 2020; Yan et al. 2020; Kondo et al. 2020), including accuracy, precision, recall, F1 score, G-Mean, ROC-AUC, etc.

In this paper, we use four of them: ROC-AUC (the area under the curve of the receiver operating characteristics), precision, recall, and F1 score. ROC-AUC is threshold-insensitive and less susceptible to imbalanced samples (Kamei et al. 2013; Kamei et al. 2016). Although precision, recall, and F1 score are threshold-based metrics, which may lead to different outcomes as the threshold changes (Lessmann et al. 2008). However, they can reflect the model performance in various aspects. It is helpful in defect prediction tasks, i.e., a higher precision score implies that fewer efforts are wasted on false positives. In comparison, a higher recall score demonstrates a higher proportion of defects are unveiled. Therefore, we use them as auxiliary metrics, whose thresholds are 0.5.

Effort-aware evaluation metrics are also widely used for defect prediction (Jiang et al. 2013; Kamei et al. 2013; Yang et al. 2015; Yang et al. 2016; Huang et al. 2017; Fu and Menzies 2017). As a means of software quality assurance, defect prediction aims to alleviate the efforts of code inspection to identify potential faults and make developers focus on the most crucial or defective part of a software project. Prior work suggested that most of the system faults are in a small percentage of files, e.g., 20 percent of files (Ostrand et al. 2005). It means that only the most defect-prone commits are inspected due to the limitation of efforts in real scenarios of JITDP. Thus, we also evaluate the cost-effectiveness of different approaches with the constraint of the effort.

We adopt two effort-aware metrics in our paper, PofB20 and P_{opt}. PofB20 refers to the percentage of the total number of bugs when only inspecting 20% of the committed LOC, i.e., LA+LD (Jiang et al. 2013). It is a normalized measure of NofB20 (Rahman et al. 2012). We choose PofB20 because the number of bugs varies significantly in different projects.

P_{opt} is another effort-aware metric proposed by Mende and Koschke (2009), which also considers the actual distribution of faults by calculating the difference between the predicted model and the optimal model. It is defined as:

$$P_{opt} = 1 - \frac{\Delta_{opt}}{AUC_{opt}}$$

$$\Delta_{opt} = AUC_{opt} - AUC_{pred}$$

where AUC_{opt} and AUC_{pred} are the area under the curve of the optimal model and that of the predicted model in the effort-based cumulative chart. As shown in Figure 3, The shaded area is Δ_{opt}. The smaller it is, the closer the predicted model is to the optimal one.

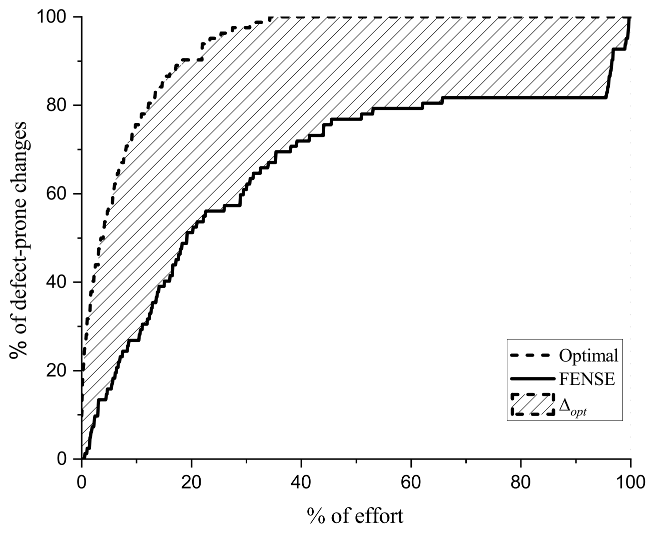


Fig. 3 Example of an effort-based cumulative chart for FENSE

For all approaches, we prioritize their prediction results by $R_d(x)$ Mende and Koschke (2009), which is defined as:

$$R_d(x) = \frac{y_i}{effort(c_i)}$$

where y_i is the predicted label of the commit c_i, and $effort(\cdot)$ is calculated by LA+LD.

Compared with data-level transfer methods, FENSE is theoretically simpler because it leverages the trained local models and does not require retraining when predicting a new target or when new data arrives. We report each method's time and space costs to demonstrate their complexity. Specifically, we split the whole application process of each cross-project approach into two phases, model training and model application. For model-level transfer approaches, we report their local model training time in a serial manner. Note that it is only required when no off-the-shelf models are provided and it also allows distribute training in practices. In our paper, we use Intel i7-9700K CPU and GeForce RTX 2080Ti GPU as our experimental environment.

4.3.3 Comparing Multiple Approaches

In this paper, we compare FENSE with five methods. Four of them are model-level transfer, including Bellwether+ and three ensemble models (random-ensemble, sim-ensemble, and all-ensemble). A model selection scheme is applied in each of these approaches, which allows us to compare the cross-project performance of their selected models. Besides, a common data-level transfer approach, data-merging, is also included in our analysis to show the effectiveness of model-level transfer in large-scale cross-project scenes.

Bellwether+ This approach is proposed by Krishna et al. (2016). 'Bellwether' is the model with the best cross-project performance in most projects. However, we cannot find bellwether in our dataset because it is difficult for a model to outperform the others on hundreds of projects. To extend the idea of the bellwether to a statistical one, we redefine it as the model that performs significantly better than the others in most projects. We name it 'Bellwether+', and it can be obtained as follows:

Given the training project set $S_{train}(P_i, P_j, P_k \in S_{train})$. We can obtain Bellwether+ by performing the following steps:

1. For each project P_i in S_{train}, use M_j (within-project model of $P_j (i \neq j)$ to predict the likelihood of defect-prone commits, whose performance is y_{ij}.
2. For project P_i and model M_j, use the Wilcoxon Rank Sum Test and Bonferroni Correction to determine whether y_{ij} is significantly better than those of other models, i.e., $y_{ik}(k \neq i, k \neq j)$.
3. Count the number of projects where M_j outperforms the others and use the best model as 'Bellwether+'.

Random-Ensemble To ascertain the effectiveness of our model selection strategy, we build the random-ensemble model for comparison. As mentioned in Section 3.2, our model is built following four steps: Pair, Predict, Select and Integrate. Random-ensemble follows a similar manner. Instead of using the fitted model to predict and rank the cross-project transferability, it randomly selects K models in S_{train} and gets a model set. Then models are integrated by average-weighted voting.

Sim-Ensemble Likewise, sim-ensemble generates a different input for ensemble learning. The idea is proposed by Kamei et al. (2016), who attempted to use project-related similarity for model selection. In our study, we measure the similarity of two projects using the Euclidean distance of project features. To be specific, sim-ensemble is built as follows:

1. **Pair:** For each project P_i in S_{train}, pair it to P_T and then extract the project metrics.
2. **Calculate:** Calculate the differences of their metrics and the Euclidean distance between P_i and P_T. A MinMaxScaler is adopted before the distance calculation.
3. **Select:** Prioritize top-K models with smallest distances. Here a model sequence is generated for the next step.
4. **Integrate:** Use selected models to predict the risk of commits for each P_T and combine them by average-weighted voting.

All-Ensemble This approach integrates models of all projects in the training set S_{train} to predict the risk of test commits in P_T. It is a general and powerful baseline in model-level transfer methods, indicating the ideal ensemble model with maximum base learners.

Data-Merging Data-merging is a common data-level transfer method, which refers to combining all the data in S_{train} to train a huge model for all cross-project JITDP tasks. It is a general and powerful baseline in data-level transfer methods, indicating the ideal model with maximum training samples.

We apply a Scott-Knott ESD test (Tantithamthavorn et al. 2017, Tantithamthavorn et al. 2018) as our techniques for the comparison test. It can cluster different approaches into ranks concerning their effect size difference. The Scott-Knott ESD test generates the ranking results according to two principles: (1) the magnitude of the difference of all distributions in each rank is negligible; (2) the magnitude of the difference of the distributions between ranks is non-negligible.

5 Results

5.1 Research Questions

In this paper, we investigate JITDP in the cross-project context by considering the characteristics of projects. Based on this contextual knowledge, we propose FENSE, an ensemble approach with a feature-based model selection scheme that selects models with higher cross-project transferability to perform JITDP. To structure our findings, we present four research questions as follows:

(RQ1) What are the factors in the cross-project transferability of JITDP models?

(RQ2) Is our feature-based model selection scheme effective? How much benefit can it bring to the cross-project performance of JITDP models?

(RQ3) Can ensemble modeling improve the cross-project performance of a single JITDP model? What are the effects of different combination methods and integration scales?

(RQ4) How does FENSE perform compared to other cross-project JITDP models?

Table 3 Summary of project features and their inter-project relations

Levels	Features	Mean	Std Dev	Min	Median	Max
Model-level	n_commits_src	4432	5708.111	312	2433	41508
	model_performance_src	0.6374	0.0635	0.5017	0.6351	0.8340
	defect_ratio_src	0.1886	0.0968	0.0439	0.1702	0.6742
Project-level	prjA_popularity	12545	13531.81	3459	7964	146808
	prjB_popularity	12545	13531.81	3459	7964	146808
	prjA_age	2747	918.46	613	2722	4736
	prjB_age	2747	918.46	613	2722	4736
	same_owner_type	0.8138	0.3893	0	1	1
	same_license	0.1083	0.3108	0	0	1
	same_language	0.3488	0.4766	0	0	1
	textual_similarity	0.0363	0.0264	0	0.0309	0.6850
Social-level	prjA_n_core	107.3	295.10	1	25	1516
	prjB_n_core	107.3	295.10	1	25	1516
	prjA_n_external	211.6	114.58	0	204	421
	prjB_n_external	211.6	114.58	0	204	421
	core_diff	167.9	382.77	0	29	1515
	external_diff	132.6	93.56	0	117	421
	contributor_intersection	0.5271	2.0862	0	0	148
	entropy_diff	0.1986	0.1487	0	0.1682	0.8526
Technical-level	prjA_code_size	197640	330071.3	2048	72774	2252135
	prjB_code_size	197640	330071.3	2048	72774	2252135
	code_size_diff	264818	385242.2	0	108299	2250087
	prjA_n_dependencies	110.7	327.16	0	4.5	1954
	prjB_n_dependencies	110.7	327.16	0	4.5	1954
	dependency_intersection	3.389	41.11	0	0	1249
	dependency_diff	196	419.84	0	20	1954

5.2 (RQ1) What are the factors in the cross-project transferability of JITDP models?

To answer RQ1, we conduct a regression analysis on a plethora of OSS in GitHub. After filtration (in Section 4.1.1 and Section 4.2.1), we finally obtain 338 projects in our dataset. For each of them, we train a local JITDP model. Then we pair each two of them and obtain 113,906 project pairs, where one's model is used to predict the risk of commits of the other. As described in Section 3.1, we extract metrics in four different levels to measure the relationship between the project pair. Their statistics are shown in Table 3. The prefixes prjA and prjB refer to the source and target projects, respectively. Note that we leave out n_core, code_size, and n_dependencies of both projects considering eliminating multicollinearity, but we keep core_diff, code_size_diff, and dependency_diff as they are more representative for inter-project relations. Finally, a linear mixed-effect model is built to investigate the impacts of project features on cross-project transferability.

Table 4 shows the result of regression analysis. The second column shows the data transformation imposed on each metric. The third and fourth column lists the coefficients and their standard errors for the fitted mixed-effect model. The fifth column shows the result of the ANOVA Type-II test. To ensure our explanation of factors is accurate, we report the conditional R^2 of our mixed-effect model, which is 0.844. It indicates a high goodness-of-fit of the model. By and large, all four

Table 4 Regression analysis of the cross-project transferability

Response: cross-project ROC-AUC

Levels	Metrics	Coeffs	Std Err	Sum Sq (Sig.)
Model-level	log(n_commits_src)	1.145e-03	1.704e-03	0.45
	log(model_performance_src)	7.827e-02	1.674e-02	21.87 ***
	log(defect_ratio_src)	3.879e-02	3.626e-03	114.45 ***
Project-level	log(prjA_popularity)	8.222e-04	2.298e-03	0.13
	log(prjB_popularity)	-1.293e-03	4.143e-03	0.10
	log(prjA_age)	1.044e-02	3.965e-03	6.94 **
	log(prjB_age)	-2.415e-02	7.022e-03	11.83 ***
	same_owner_type TRUE	1.157e-03	4.359e-04	7.04 **
	same_license TRUE	1.704e-04	3.129e-04	0.30
	same_language TRUE	2.518e-03	1.808e-04	193.91 ***
	log(textual_similarity+0.5)	5.123e-03	2.492e-03	4.23 *
Social-level	log(prjA_n_external+0.5)	6.346e-03	1.713e-03	13.73 ***
	log(prjB_n_external+0.5)	-8.835e-05	3.072e-03	0.00
	log(core_diff+0.5)	5.673e-05	8.653e-05	0.43
	log(external_diff+0.5)	-4.531e-05	7.652e-05	0.35
	log(contributor_intersection+0.5)	4.501e-05	1.429e-04	0.10
	log(entropy_diff+0.5)	-2.450e-03	5.328e-04	22.73 ***
Technical-level	log(code_size_diff+0.5)	-2.404e-04	7.363e-05	10.66 **
	log(dependency_intersection+0.5)	2.419e-04	1.388e-04	3.04 .
	log(dependency_diff+0.5)	-3.290e-04	7.841e-05	0.18
	Conditional R^2		0.844	

Significance of χ^2: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

levels more or less explain the cross-project transferability of a JITDP model, of which same_language and defect_ratio_src are the most influential metrics. In the following, we will analyze the effect of each metric elaborately.

For model-level features, two of three metrics significantly affect cross-project transferability, and their coefficients are all positive. For defect_ratio_src, its sum of squares is 114.45. Its significance indicates that the higher the proportion of defect-prone commits is, the better its trained model performs in other projects. As for model_performance_src, if a model has a higher within-project ROC-AUC value, it is more likely to behave the same in the cross-project context, which contradicts the findings in Kamei et al. (2016). The effect of n_commits_src is counterintuitive. The number of training samples has no significant influence on the cross-project performance. The result implies that we cannot improve model transferability only by increasing the local training data.

For project-level measures, same_language is the strongest predictor for cross-project prediction, whose $SumSq$ is 193.31 and explains 47% of the variance of the whole model. If the source and target projects are developed using the same programming language, their model cross-project transferability is higher than those in different languages. We hold that projects with the same programming language have similar grammatical structures, and thus their change metrics are in similar distributions. Even though our dataset only includes projects in 3 different languages, it reflects that the cross-language predicting performance is relatively low. Hence, we should prioritize models by their languages. Apart from this, the project age of the target project has a significant influence on cross-project transferability. Its coefficient is negative, which means that the performance of cross-project prediction will drop when the age of the target project gets longer. We infer that projects have more distinct features as time goes on, thereby more difficult for other models to predict the risk of their commits. Moreover, it is inappropriate to use cross-project models for projects with a long history. In this scenario, within-project models are better choices if they are available. Same_owner_type, prjA_age, and textual_similarity are also significant, whose effects are all positive. It means that the same type of management groups, long age of the source project, and similar description can enhance the cross-project transferability. Other metrics such as project popularity and license barely impact the cross-project transferability.

For social-level measures, there are two significant predictors, entropy_diff and prjA_n_external. The former is the most influential, and its coefficient is negative. It means that similar distribution of developers' contributions between two projects is essential for model transferability. We suspect that their development histories are close to each other. Hence the change metrics are also in similar distribution. Interestingly, the number of external contributors of the source project strongly impacts the transferability of prediction models. We suspect the reason is that a broader range of developers generate the development data, and thus the model can generalize better in a cross-project context.

Only two measures at the technical-level are statistically significant, code_size_diff and dependency_intersection, which both exert a negative influence on cross-project prediction. When the difference between the project size is smaller, the cross-project transferability is higher. Likewise, more common dependencies in the source and target projects indicate better cross-project predicting performance. Hence, we suggest selecting the model whose project has a similar size and more common dependencies with the target. However, the number of dependencies do not affect the cross-project JITDP.

Answer for RQ1: *In four levels of software features, many of them significantly influence the cross-project transferability of JITDP models. The most powerful one is the programming language, which explains 47% variance of the whole model. The defect ratio and local model performance at the model level, the age of the target project at the project level, the number of external contributors, and the difference of contribution entropy at the social level also have substantial impacts on the cross-project predicting performance. We recommend considering these expressive software features when selecting cross-project JITDP models.*

5.3 (RQ2) Is our feature-based model selection scheme effective? How much benefit can it bring to the cross-project performance of JITDP models?

In previous studies, a single JITDP model is believed to behave poorly in a cross-project context (Zimmermann et al. 2009). However, Krishna et al. (2016) suggested that there is always a 'bellwether' in a project community. This 'bellwether' is considerably simple and can even outperform some data-level transfer methods. However, Krishna contended that statistics cannot predict it unless tested against other datasets. Leveraging the result of RQ1, we believe that the con-

Table 5 The average cross-project performance of the best models selected by different schemes

Methods	ROC-AUC	Precision	Recall	F1	PofB20	P_{opt}
Bellwether+	0.6770	0.3201	0.5168	**0.3654**	0.3447	0.5352
Random-ensemble	0.6250	**0.3677**	0.3567	0.3047	0.2773	0.5088
Sim-ensemble	0.6299	0.3644	0.3670	0.3170	0.2834	0.5137
FENSE-1	**0.6854**	0.2596	**0.6458**	0.3475	**0.4178**	**0.5962**

* FENSE-1 means only the top model is selected for evaluation

textual knowledge about software features can contribute to the identification of powerful cross-project models. Hence, we propose a feature-based model selection scheme to identify candidates with higher cross-project performance. To validate our method, we obtain the cross-project performance of our selected models and compare the model selection scheme of FENSE with that of Bellwether+, random-ensemble, and sim-ensemble.

To avoid data leakage from test projects, we use a stratified sampling approach to pick out an isolated test set in different programming languages. We randomly select 67 projects (16 for Java, 28 for C/C++, and 23 for Python) as the test project set S_{test}, accounting for 20% of each language. This train-test-split of our dataset is replicated ten times to mitigate the influence of randomization. Note that a new mixed-effect model is built relying on different training data.

In Section 3.2 and Section 4.3.3, we introduce the model selection schemes of different approaches. FENSE builds a mixed-effect model to predict and rank the cross-project transferability of models on each target project, while sim-ensemble ranks models by the similarity of project features. As for baselines, we choose Bellwether+ and random-ensemble. The former represents the model with the highest generalization ability among the training projects. The latter randomly selects a group of models from the training projects and represents the average level of cross-project performance. The random selection is also repeated ten times.

Table 5 shows the average cross-project performance of the best models selected by different approaches. Among the six measures, the ROC-AUC value is our primary focus. The reason is that we adopt the ROC-AUC value as our response variable in regression analysis and the criterion for Bellwether+ to measure the cross-project transferability. It directly reflects the excellence of the model selection scheme.

The result illustrates that FENSE can select the model with the highest ROC-AUC score, which improves by nearly 10% compared to random-ensemble (without any model selection operations). It also outperforms Bellwether+ slightly. However, its improvement of ROC-AUC is numerically marginal. It is probably related to the distribution of the cross-project JITDP performance, with a Mean(Std) of 0.6255(0.0671). Hence, we also report the ranks of FENSE and Bellwether+ on ROC-AUC in Table 6.

The median rank of FENSE on ROC-AUC is 18 (among 271 training projects), which shows that FENSE is capable of selecting the model in the top 6.6%. As for Bellwether+, it only selects one model to predict the defect-proneness of all test projects, so its rank is below the level of the top 10%. To better illustrate the strength of FENSE, we also perform the Mann-Whitney U test for the scores and

Table 6 The median ranks of FENSE-1 and Bellwether+ on ROC-AUC among all training projects

Methods	Rank
Bellwether+	27.5
FENSE-1	18.0 ***

Signif: 0 '***' 0.001

Table 7 The results of Mann-Whitney U test for FENSE-1 and Bellwether+

Methods	Bellwether+	FENSE-1
ROC-AUC	0.6770	**0.6854** **
Precision	0.3201 ***	0.2596
Recall	0.5168	**0.6458** ***
F1	**0.3654** **	0.3475
PofB20	0.3447	**0.4178** ***
P_{opt}	0.5352	**0.5962** ***

Signif: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

ranks between these two approaches. The results are shown in Table 6 and Table 7.

The advantage of FENSE in ROC-AUC is statistically significant (p<0.01). Moreover, FENSE also performs best in effort-aware scenarios because FENSE considers project features' variance, thereby finding suitable models according to different target projects. However, it is weaker than Bellwether+ with respect to precision and F1 score.

Note that the model selected by FENSE is characterized by its high recall and low precision score, 0.6458 and 0.2596, respectively. In other words, this model can retrieve two-thirds of the defect-prone commits. However, there are nearly three false positives for each real buggy one. We will further analyze this phenomenon in RQ3 and RQ4 when more models are selected and integrated.

As for sim-ensemble, it has nearly the same ROC-AUC score as random-ensemble, which means that the similarity of the project metrics is not a good indicator for cross-project transferability. To verify our judgment, we also evaluate the model sequences generated by FENSE and sim-ensemble. Both of them rank the JITDP models and produce a sequence for subsequent ensemble modeling. If an approach can identify better models in a cross-project context, its sequence must conform to the actual cross-performance of models. Thus, we calculate their Normalized Discounted Cumulative Gain (NDCG). NDCG is a measure of ranking quality, which is often used to evaluate recommendation systems (Jarvalin et al. 2002). In this paper, we assign higher relevance scores for models with higher cross-project performance and obtain results in Table 8. The values are the mean scores on 67 test projects in ten repeat studies. K means the sequence comprises the top-K models selected by each method.

We choose the sequence length as 1, 5, 10, 20, and 50 to evaluate the top models preferred by each method. Random-ensemble is also used as a baseline. The result confirms our judgment. FENSE is able to rank the cross-project transferability of models by considering project features comprehensively, while similarity cannot

Table 8 NDCG-K of the model sequences selected by FENSE, random-ensemble and sim-ensemble

Methods	NDCG-1	NDCG-5	NDCG-10	NDCG-20	NDCG-50
Random	0.5280	0.5350	0.5374	0.5426	0.5642
Similarity	0.5188	0.4966	0.5076	0.5225	0.5458
FENSE	**0.8853**	**0.8816**	**0.8803**	**0.8831**	**0.8954**

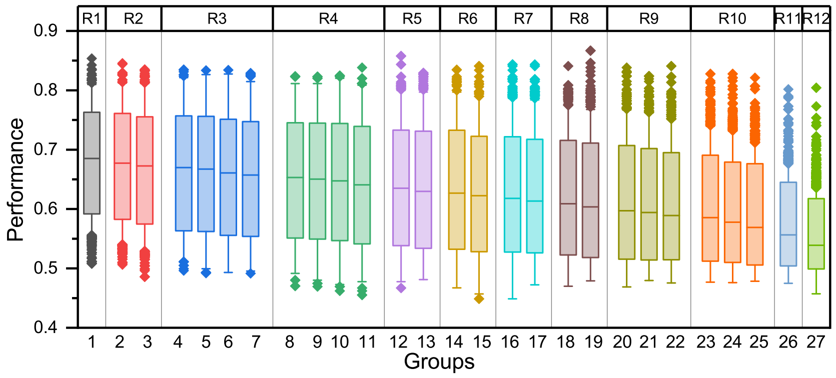


Fig. 4 The actual ROC-AUC scores of the model sequence generated by FENSE

cope with their relationship. This is why Kamei et al. (2016) cannot improve the ensemble model by considering project similarity.

In addition, we calculate the ROC-AUC scores of models on the test set to validate whether our model selection scheme can actually rank the models by its estimation. We divide the model sequence into groups according to their ranks. Each group contains ten models, and we apply Scott-Knott ESD test to these groups. The result is shown in Figure 4, which indicates that our feature-based model selection scheme is effective in prioritizing highly transferable models. The x-axis is the number of the group followed the order of estimated performance.

Answer for RQ2: *Our feature-based model selection scheme effectively selects JITDP models with higher cross-project transferability (measured by ROC-AUC). Compared to random-ensemble, it can improve nearly 10% and even outperform Bellwether+ when selecting a single model. In contrast, the similarity of project metrics cannot capture the relationship of our contextual knowledge and the cross-project performance of JITDP models. We suggest considering the effects of project features comprehensively as we did in RQ1.*

5.4 (RQ3) Can ensemble modeling improve the cross-project performance of a single JITDP model? What are the effects of different combination methods and integration scales?

Ensemble modeling is a common strategy to enhance cross-project JITDP performance. Instead of finding a powerful learner, it turns to the combination of multiple models to achieve better cross-project prediction, and the combination method thereof is essential (Zhou 2012). Based on the findings of RQ2, we evaluate the impacts of ensemble modeling with different combination methods and integration scales.

We build ensemble models using four different combination methods. One is majority voting, and the other three belong to soft voting with different weights. The majority voting is the most popular combination method for classification tasks. For JITDP, its output is

$$E(\mathbf{c}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{K} f_i(\mathbf{c}) > \frac{1}{2}K, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

where $\mathbf{c}$ is the metrics of a commit, $f_i(\cdot)$ is the result of i-th base learner (0 or 1) and K is the number of combined models.

As for soft voting, it combines the class probability outputs of individual models. A classifier-specific weight can be assigned to each classifier. Hence the output is

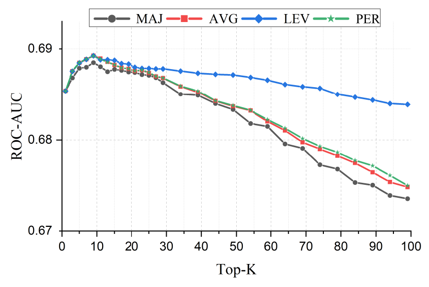
$$E(\mathbf{c}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{K} w_i p_i(\mathbf{c}) > 0.5, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

where w_i is the weight of i-th base learner. w_i is positive and satisfies $\sum_{i=1}^{K} w_i = 1$. $p_i(\cdot)$ is the defect-proneness of i-th base learner. In our study, we use Random Forest as the base learner and p_i is the percentage of decision trees that predict $\mathbf{c}$ as buggy.

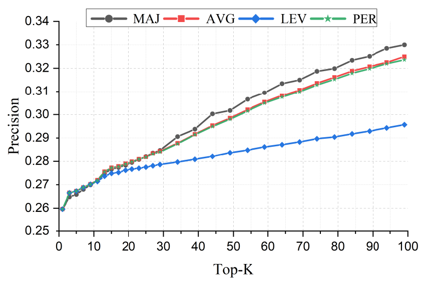
In RQ2, we validate the effectiveness of the model sequence produced by our feature-based selection scheme. It is in the performance order. Hence we assume that giving models with better performance higher weights can enhance ensemble learning. In this paper, the weights are given in three different manners.

- **Average**:
$$w_i = \frac{1}{K}$$
Models have same weights.
- **Multilevel**:
$$\omega_i = \frac{1}{2^l} \quad l = \lfloor i/n \rfloor$$
$$w_i = \frac{\omega_i}{\sum_{j=1}^{K} \omega_j}$$
n is the number of models at the same level. As RQ2 suggests, the cross-project performance of the model decreases in the sequence when i rises. Hence, we adjust the weight of a base learner by the level of its estimated performance. In this study, we set n as 10.
- **Performance**:
$$\omega_i = \frac{1}{1 - y_i}$$
$$w_i = \frac{\omega_i}{\sum_{j=1}^{K} \omega_j}$$
y_i is the cross-project performance of i-th base learner. In our study, it is the ROC-AUC score. Then models are combined by their estimated performance.

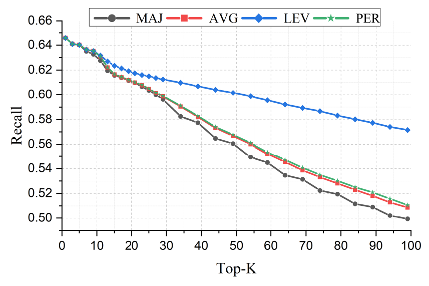
To answer RQ3, we obtain the performance of different ensemble models as the hyperparameter K grows. We set $K = 1$ initially and increase its value by 2 when it is less than thirty and by 5 later on. Similar to RQ2, the train-test-split of our dataset is replicated ten times, and the reported performance on the test set are their average values. The results are shown in Figure 5.



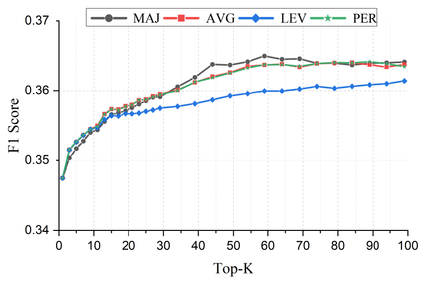
(a) ROC-AUC



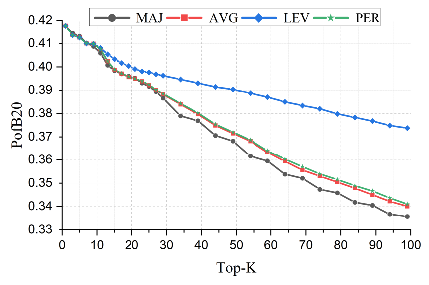
(b) Precision



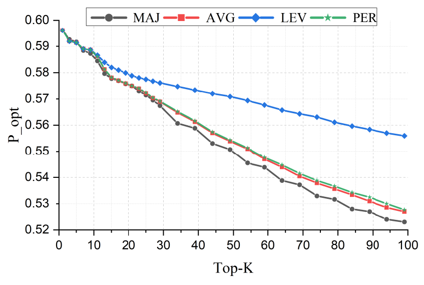
(c) Recall



(d) F1 score



(e) PofB20



(f) P_{opt}

Fig. 5 Model performance of four combination methods as the integration scale grows

The curves show that all four combination methods have the same behaviors when K rises, and their differences are not significant. Ensemble modeling with majority voting performs slightly worse than soft voting methods except for precision values. Average-weighted and performance-weighted methods are almost the same on all evaluation metrics, which suggests that a small gap between model performance makes no difference in ensemble modeling. The behavior of the multilevel-weighted method stands out when K increases. Because the late-comers

Table 9 The integration scale of ensemble modeling

K	Brief Description
1	Single model
5	Small scale ensemble (a)
9	Small scale ensemble (b)
19	Medium scale ensemble
39	Large scale ensemble
99	Extra large scale ensemble
271	Full scale (all-ensemble)

have a lower weight, this approach tends to retain the properties of the top models, i.e., high ROC-AUC and low precision scores.

To deeply study the effect of ensemble modeling, we consider several representative K and analyze the impact of integration scale qualitatively. Table 9 shows the levels of integration scale in this paper, and Table 10 is the summary of cross-project performance for ensemble models with four different combination methods in different integration scales. The results of random-ensemble and sim-ensemble are also provided to compare the impacts of feature-based model selection to ensemble modeling without this contextual knowledge.

Firstly, the effect of ensemble modeling on FENSE is the same whichever combination method is taken. It improves the ROC-AUC when the integration scale is small. All four methods reach their best when $K = 9$. However, the ROC-AUC score falls when the scale continues to increase, which is probably because the input of our ensemble model is a sequence with decreasing ROC-AUC scores. As for precision and F1 score, their values grow as the integration scale rises, which suggests that the combination of more models can benefit the classification of clean commits. However, the recall scores become lower for all methods, including random-ensemble and sim-ensemble, because it is harder for models to distinguish defective commits in cross-project context than clean ones. In addition, from the effort-aware perspective, all four ensemble models are less cost effective when K rises. Taking FENSE-AVG-99 as an example, we could discover 18.6% fewer defects than FENSE-AVG-1 with the same code inspection cost. To summarize, there are only slight differences in ROC-AUC, precision and F1 score when the integration scale changes, while the recall, PofB20 and P_{opt} of FENSE significantly decrease when the integration scale is extra large, i.e., $K = 99$.

Secondly, the effects of ensemble modeling on random-ensemble and sim-ensemble are different from above. How their performance change with the hyperparameter K is shown in Figure 6. For sim-ensemble, it obtains its best performance on 3 of 4 non-effort-aware metrics when $K = 19$, while random-ensemble have better non-effort-aware performance when the integration scale is larger. On the contrary, these two approaches achieve worse effort-aware performance when K grows. Their largest performance decrease happens at the beginning of the integration. Similar to FENSE, ensemble modeling gives no aids to effort-aware cross-project JITDP.

Thirdly, we report the time and space costs of FENSE and sim-ensemble when K increases in Table 11 and Figure 7. Here we only consider the costs during model prediction phase to investigate the effectiveness in different integration sizes.

We can see from the results that the time costs of both models increase linearly with K. FENSE consumes nearly 50% less memory than sim-ensemble when $K <$

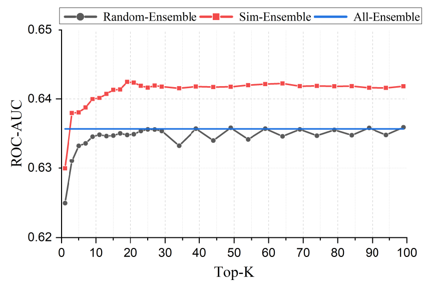
Table 10 Summary of cross-project performance for ensemble models with different combination methods and in different scales

Metrics	Methods	K					
		1	5	9	19	39	99
ROC-AUC	MAJ	**0.6854**	0.6879	0.6885+	0.6875	0.6849	0.6736
	AVG	**0.6854**	**0.6885**	**0.6893**+	0.6878	0.6852	0.6749
	LEV	**0.6854**	**0.6885**	**0.6893**+	**0.6883**	**0.6873**	**0.6839**
	PER	**0.6854**	**0.6885**	0.6892+	0.6879	0.6854	0.6750
	RAN	0.6250	0.6333	0.6346	0.6349	0.6357	0.6359+
	SIM	0.6300	0.6381	0.6400	0.6425+	0.6418	0.6419
Precision	MAJ	0.2596	0.2659	0.2700	0.2784	0.2938	0.3300+
	AVG	0.2596	0.2673	0.2702	0.2790	0.2916	0.3250+
	LEV	0.2596	0.2673	0.2702	0.2762	0.2809	0.2957+
	PER	0.2596	0.2674	0.2701	0.2788	0.2914	0.3236+
	RAN	**0.3677**	**0.3971**	**0.4042**	**0.4094**	**0.4137**	**0.4154**+
	SIM	0.3644	0.3915	0.3983	0.4039+	0.402	0.4026
Recall	MAJ	**0.6458**+	**0.6404**	0.6328	0.6117	0.5774	0.4994
	AVG	**0.6458**+	0.6402	**0.6354**	0.6119	0.5821	0.5086
	LEV	**0.6458**+	0.6402	**0.6354**	**0.6192**	**0.6070**	**0.5717**
	PER	**0.6458**+	0.6402	0.6353	0.6121	0.5829	0.5106
	RAN	0.3567+	0.3501	0.3487	0.3461	0.3467	0.3462
	SIM	0.3670+	0.3626	0.3622	0.3639	0.3622	0.3617
F1	MAJ	**0.3475**	0.3518	0.3540	0.3571	**0.3619**	**0.3641**+
	AVG	**0.3475**	**0.3527**	**0.3545**	**0.3578**	0.3612	0.3637+
	LEV	**0.3475**	**0.3527**	**0.3545**	0.3567	0.3582	0.3614+
	PER	**0.3475**	**0.3527**	**0.3545**	0.3577	0.3612	0.3634+
	RAN	0.3047	0.3285	0.3326	0.3347	0.3369	0.3380+
	SIM	0.3170	0.3399	0.3456	0.3511+	0.3488	0.3487
PofB20	MAJ	**0.4178+**	**0.4132**	0.4088	0.3960	0.3770	0.3357
	AVG	**0.4178+**	0.4126	**0.4099**	0.3956	0.3798	0.3401
	LEV	**0.4178+**	0.4126	**0.4099**	**0.4004**	**0.3931**	**0.3738**
	PER	**0.4178+**	0.4126	**0.4099**	0.3958	0.3803	0.3410
	RAN	0.2773+	0.2645	0.2627	0.2611	0.2603	0.2604
	SIM	0.2834+	0.2655	0.2644	0.2638	0.2614	0.2603
P_{opt}	MAJ	**0.5962**+	**0.5917**	0.5873	0.5758	0.5589	0.5231
	AVG	**0.5962**+	0.5914	**0.5887**	0.5759	0.5613	0.5270
	LEV	**0.5962**+	0.5914	**0.5887**	**0.5799**	**0.5734**	**0.5560**
	PER	**0.5962**+	0.5914	0.5886	0.5760	0.5617	0.5277
	RAN	0.5088+	0.4838	0.4798	0.4769	0.4752	0.4739
	SIM	0.5137+	0.4857	0.4796	0.4762	0.4730	0.4709

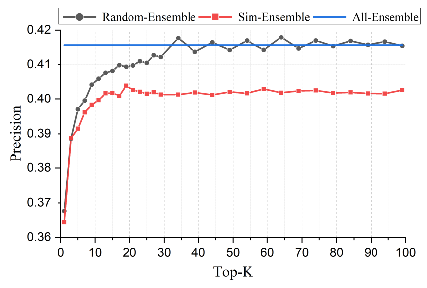
Bold text means an approach achieves the best performance among all methods;
'+' symbol means an approach achieves the best performance among all integration scales.

20, but their values remain steady when $K > 30$. If an ensemble method achieves a competitive results when K is small, we had better not integrate more models because of its cost and decreasing performance, particularly effort-aware ones.

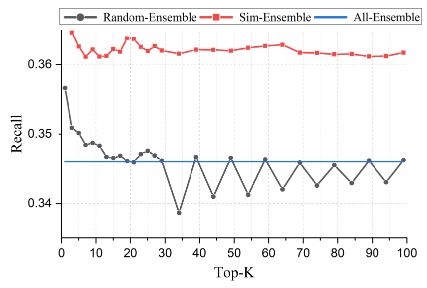
Note that when $K = 271$, both FENSE and random-selection will degrade to all-ensemble. FENSE still has its distinct characteristics when the integration scale is extra-large. However, random-selection converges to all-ensemble quickly, which can be derived from Figure 5 and Figure 6. Nevertheless, the comparison between FENSE and the other two approaches suggests that merely integrating more mod-



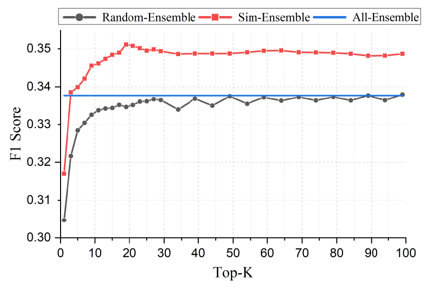
(a) ROC-AUC



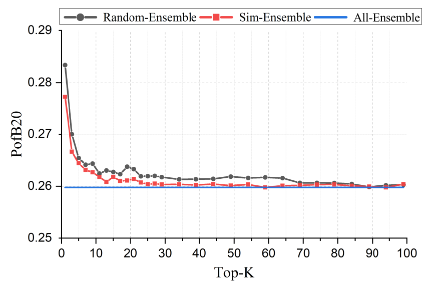
(b) Precision



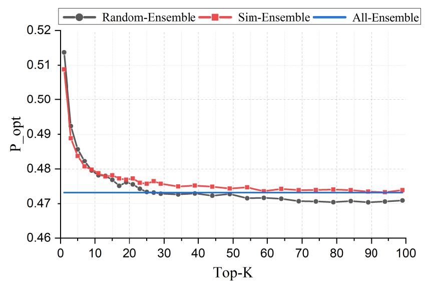
(c) Recall



(d) F1 score



(e) PofB20



(f) P_{opt}

Fig. 6 The performance of random-ensemble and sim-ensemble as the integration scale grows

els without proper consideration of software features cannot benefit cross-project JITDP.

Answer for RQ3: *For FENSE, the effect of ensemble modeling on cross-project JITDP is not comparable to that of the model selection scheme. Combination methods and integration scales are both less influential for FENSE. However, the integration scale benefits random-ensemble and sim-ensemble in non-effort-aware scenarios. When it comes to cost-effectiveness, ensemble modeling provides no supports for cross-project JITDP, no matter which approach is used. Considering the costs of model application, ensemble approaches should only integrate several models rather than hundreds of them if their differences are not significant.*

Table 11 Time and space costs of FENSE and sim-ensemble during model prediction phase in different integration scales

Methods	Costs	K					
		1	5	9	19	39	99
FENSE	Space	289.22	311.01	299.29	301.29	663.84	666.03
	Time	9.89	41.77	70.71	136.73	283.46	688.77
sim-ensemble	Space	368.15	512.79	564.78	569.26	593.82	661.97
	Time	9.61	38.18	66.10	134.92	276.49	700.18

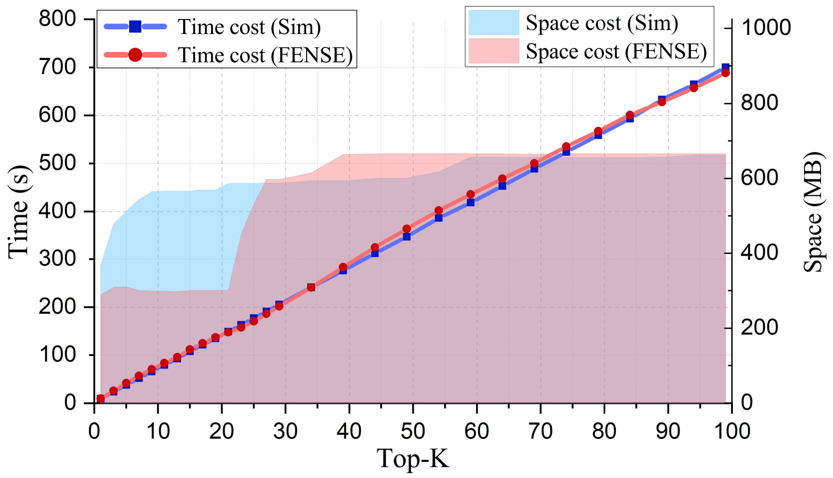


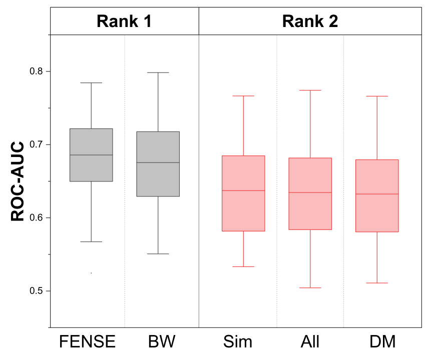
Fig. 7 Time and space costs of FENSE and sim-ensemble during model prediction phase when K increases

5.5 (RQ4) How does FENSE perform compared to other cross-project JITDP models?

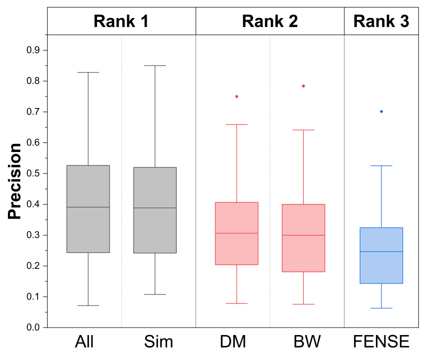
To evaluate the performance of FENSE more elaborately, we compare it to the other four common cross-project methods and apply the Scott-Knott ESD test to the prediction results on six different evaluation metrics. Similar to RQ2 and RQ3, the results are the mean scores on 67 test projects in 10 replicated studies. In particular, we use average-weighted combination methods and $K = 9$ for FENSE while adopting average-weighted and $K = 19$ for sim-ensemble to represent their performance better, leveraging the results from RQ3.

Table 12 The average cross-project JITDP performance of five approaches

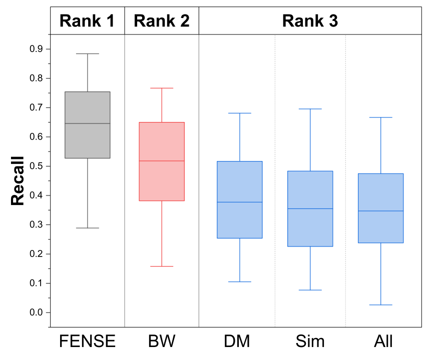
Methods	ROC-AUC	Precision	Recall	F1	PofB20	P_{opt}
Data-Merging	0.6352	0.3334	0.3749	0.3309	0.2888	0.5149
Bellwether+	0.6770	0.3201	0.5168	**0.3654**	0.3447	0.5352
Sim-Ensemble-19	0.6425	0.4039	0.3639	0.3511	0.2638	0.4762
All-Ensemble	0.6357	**0.4156**	0.3460	0.3377	0.2598	0.4732
FENSE-AVG-9	**0.6893**	0.2702	**0.6354**	0.3545	**0.4099**	**0.5887**



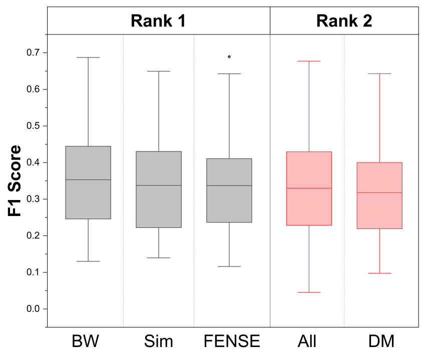
(a) Scott-Knott ESD test on ROC-AUC



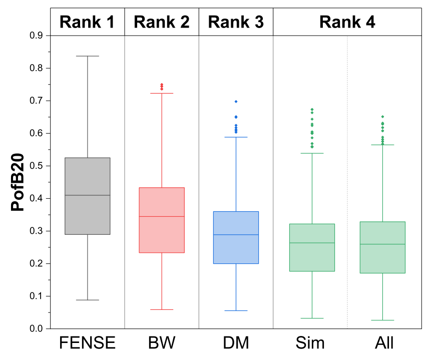
(b) Scott-Knott ESD test on precision



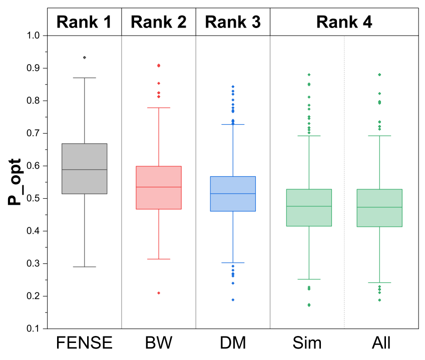
(c) Scott-Knott ESD test on recall



(d) Scott-Knott ESD test on F1 score



(e) Scott-Knott ESD test on PofB20



(f) Scott-Knott ESD test on P_{opt}

Fig. 8 The results of Scott-Knott ESD test on six evaluation metrics (BW is the abbreviation for Bellwether+, while DM for data-merging, Sim for sim-ensemble and All for all-ensemble)

Figure 12 and Table 8 illustrate our experimental results. For the ROC-AUC score, FENSE and Bellwether+ have an apparent advantage over sim-ensemble, all-ensemble and data-merging. FENSE gets 0.6893, and Bellwether+ gets 0.677. For precision, all-ensemble gets 0.4156 and performs best, followed by sim-ensemble with a score of 0.4039. FENSE ranks at the third level with only 0.2702 in pre-

Table 13 The results of Mann-Whitney U test for FENSE-AVG-9 and Bellwether+

Methods	Bellwether+	FENSE-AVG-9
ROC-AUC	0.6770	**0.6893** ***
Precision	0.3201 ***	0.2702
Recall	0.5168	**0.6354** ***
F1	**0.3654** .	0.3545
PofB20	0.3447	**0.4099** ***
P_{opt}	0.5352	**0.5887** ***

Signif: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 14 Time and space costs of five methods

Methods	Training		Application	
	Space (MB)	Time (s)	Space (MB)	Time (s)
Data-Merging	7281.71	7048.49	18538.97	204.52
Bellwether+	425.33	3895.76	189.92	44.91
Sim-Ensemble-19	425.33	3895.76	569.26	342.05
All-Ensemble	425.33	3895.76	573.01	1824.81
FENSE-AVG-9	425.33	3895.76	940.20	285.63

cision. Nevertheless, the recall score of all approaches behaves oppositely against precision. FENSE achieves 0.6354 while Bellwether+ only gets 0.5168 in the second rank. The rest approaches only have recall scores below 0.4. With respect to the F1 score, Bellwether+, sim-ensemble, and FENSE are in the first rank.

From the effort-aware aspect, FENSE are the best approach with a PofB20 of 0.4099 and a P_{opt} of 0.5887, which means that developers can identify 40% of all defect-prone commits when only inspecting 20% committed LOC. Bellwether+ ranks second. Its PofB20 still outperforms other approaches, while it has no obvious advantage on P_{opt}. Data-merging is less effective than FENSE and Bellwether+, followed by sim-ensemble and all-ensemble.

According to the Scott-Knott ESD test, FENSE and Bellwether+ both rank first on ROC-AUC and F1 score. To further investigate whether their differences are significant, we employ the Mann-Whitney U test for their performance.

As illustrated in Table 13, FENSE outperforms Bellwether+ significantly with respect to the ROC-AUC score, recall, PofB20 and P_{opt} ($p<0.001$), while their difference in the F1 score is not significant ($p>0.05$). The result demonstrates that FENSE is a better approach to the cross-project JITDP task than Bellwether+.

Additionally, we report the time and space costs for all approaches in Table 14. In the left two columns, we provide the average model training time in 10 repeated studies. For model-level transfer approaches, we report the training time for 271 local training projects in a serial manner. Note that it will cost nothing if local models have been trained. And it allows distribute training in real scenarios. Likewise, in the right two columns, we provide the total prediction time for 67 test projects and average them in 10 repeated studies. For FENSE, its costs in different application phases is also compared with sim-ensemble in Table 15.

Data-merging requires the most time and memory usage when training among the five approaches because it builds a JITDP learner using millions of training samples. Its model application also has a large memory cost, however, its perfor-

Table 15 Time and space costs of FENSE and sim-ensemble in different application phases

Methods	Phases					
	Feature Preparation		Model Selection		Model Prediction	
	Space	Time	Space	Time	Space	Time
Sim-Ensemble-19	303.01	205.85	234.92	1.28	569.26	134.92
FENSE-AVG-9	303.01	205.85	940.20	9.07	299.29	70.71

Space is measured in MB, time is measured in s.

mance is far from satisfactory. In contrast, model-level transfer methods cost less and its models also allow distribute training, which is more practical and efficient in real scenarios.

For Bellwether+, it is also competitive with respect to the model costs, even though a large number of projects are provided. As we suggested before, it uses the model with the highest generalization ability to predict target projects. Its good performance also indicates a fair cross-project predictor in our dataset. From our statistics, it is a popular Python repository Scrapy[5], whose JITDP model outperforms 97% models in the training set in cross-project context. It is worth going further to discover why these bellwethers are such outstanding.

For all-ensemble, its time cost is much more higher than others but it has the worst result except precision, which demonstrates that merely increasing the number of integrated models provides no good for cross-project JITDP.

As for sim-ensemble, it has similar time cost but less memory usage compared to FENSE. The extra space cost origins from the regression model of FENSE. However, without this analysis, sim-ensemble cannot handle the effects of project features on cross-project transferability, thus obtains a weak performance on most metrics.

For our proposed FENSE, it ranks first on 5 out of 6 measures except precision. The result highlights its ability to retrieve defect-prone commits and advantages in effort-aware scenarios. Combined with the results from RQ2 and RQ3, we find that our scheme tends to prioritize 'radical' models that are more likely to classify commits as buggy ones. Moreover, as the integration scale (or K) grows, its F1 score rises but falls in PofB20 and P_{opt}. Importantly, the cost of our feature-based model selection scheme is acceptable. For each test project, the feature preparation and model selection only costs $215/67 = 3.2s$. The peak memory usage in the whole process is less than 1GB.

Answer for RQ4: *FENSE ranks first on 5 out of 6 measures compared to existing cross-project approaches. Its result also highlights that our feature-based selection scheme is powerful to retrieve more defect-prone commits and is more cost-effective. Bellwether+ performs well in our dataset. The model trained on a popular Python project outperforms 97% models in cross-project context. The reason is worth further investigation. Additionally, the costs of feature-based model selection scheme in FENSE is negligible compared to data-level transfer approaches.*

[5] https://github.com/scrapy/scrapy/

6 Threats to Validity

To manifest the construct validity of our study, we use six evaluation metrics (ROC-AUC, precision, recall, F1 score, PofB20 and P_{opt}) to evaluate the effectiveness of the defect prediction model, which is a common measure used in classification problems. Moreover, for the features proposed in this paper, we conduct a statistical test to our mixed-effect model to ascertain their influences. Finally, performance among methods is checked using the Mann-Whitney U test and the Scott-Knott ESD test. As for internal validity, we only consider the numerical values of features. Their distributions may also have impacts on the cross-project transferability. When building local JITDP models, we remove metrics from the History and Experience aspects because they are not accessible for new projects. However, these metrics could positively affect the model performance as our RQ1 result suggests. Thus, taking them into consideration can further improve our models. Normalization is required to make the metrics less project-specific. During data annotation, we use an improved SZZ algorithm to annotate the changes. However, it may still introduce noises that cause the low prediction performance of models. For instance, the effects of tangled commits and undetected non-functional changes. Prior studies show that the mislabel changes do not influence our major findings (Fan et al. 2019). Furthermore, we repeat our experiments to eliminate the impact of randomness introduced in our study. As for external validity, we conduct our analysis using a large scale of data from OSS projects in GitHub. But it is still unclear if our approach can also perform well in commercial projects or OSS projects hosted on other platforms. What is more, we choose Random Forest as our base learner and whether our regression model can predict other learners' transferability is still a question. And the judgments on our ensemble approach are not guaranteed in other prediction tasks. As for the comparisons among methods, the state-of-the-art models should be taken into consideration, such as deep-learning-based approaches.

7 Conclusion

Data-driven methods like machine learning improve the intelligence in the software domain, but they do not work when historical data is not available. Cross-project approaches are proposed to address this problem, which assumes that different projects share similar defect-related features. However, projects have various characteristics in JITDP tasks, and a cross-project model is unlikely to perform well without proper consideration. This paper proposes FENSE, a feature-based ensemble modeling approach to cross-project JITDP. Its model selection scheme considers the variance of project features and integrates models with higher transferability. To support this, we conduct a large-scale empirical study on 113,906 pairs built on 338 OSS projects in GitHub. We identify several influential metrics for cross-project performance, such as programming language and the defect ratio of the source project. In addition, we analyze the effects of our model selection scheme and ensemble modeling. For ensemble models, different combinations and integration scales are investigated. Finally, we validate FENSE on an isolated test set by comparing it to other cross-project JITDP approaches on six evaluation metrics. We draw the following conclusions:

- We prove that project features play an important role in cross-project JITDP, especially the programming language and the defect ratio of the source project. However, a suitable model selection scheme is required to consider their impacts comprehensively, i.e., the similarity of project metrics cannot capture the relationship between our context knowledge and the cross-project transferability.
- The effect of ensemble modeling is not comparable to the model selection scheme. Ensemble models can achieve their expected performance by integrating several models, and merely integrating more models without proper consideration of software features cannot benefit cross-project JITDP.
- Our proposed FENSE outperforms other cross-project approaches on 5 out of 6 metrics. It tends to prioritize 'radical' models that are more likely to classify commits as buggy ones, which can retrieve more defect-prone commits. Apart from this, FENSE is more cost-effective than other approaches considering the inspection process. It is also resource-saving and allows distributed training.
- Similar to Krishna et al. (2016), our study demonstrates that the idea of model-level reusing is encouraging because both Bellwether+ and FENSE can select a single model with high performance in a cross-project context. However, their identification relies on the scale of the study. We suggest that future research on cross-project approaches should be conducted on a large dataset to discover the factors in transferability and better evaluate them.

In future work, we will investigate how to utilize more knowledge in software development and maintenance to support cross-project JITDP, such as the semantics of changes. Moreover, our results will be compared with modern deep learning approaches that attain state-of-the-art performance on various software tasks. Model effectiveness also needs to be validated in real scenarios to give aids to software practices.

References

Aversano L, Cerulo L, Del Grosso C (2007) Learning from bug-introducing changes to prevent fault prone code. In: Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting, Association for Computing Machinery, New York, NY, USA, IWPSE '07, p 19–26, DOI 10.1145/1294948.1294954, URL https://doi.org/10.1145/1294948.1294954

Bettenburg N, Hassan AE (2010) Studying the Impact of Social Structures on Software Quality. In: 2010 IEEE 18th International Conference on Program Comprehension, IEEE, Braga, Portugal, pp 124–133, DOI 10.1109/ICPC.2010.46, URL http://ieeexplore.ieee.org/document/5521754/

Briand L, Melo W, Wust J (2002) Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Transactions on Software Engineering 28(7):706–720, DOI 10.1109/TSE.2002.1019484

Cabral GG, Minku LL, Shihab E, Mujahid S (2019) Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction. In: Proceedings - International Conference on Software Engineering, IEEE, vol 2019-May, pp 666–676, DOI 10.1109/ICSE.2019.00076

Capiluppi A, Lago P, Morisio M (2003) Characteristics of open source projects. In: Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings., pp 317–327, DOI 10.1109/CSMR.2003.1192440

Catolino G, Di Nucci D, Ferrucci F (2019) Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In: Proceedings of the 6th International Conference on Mobile Software Engineering and Systems, IEEE Press, MOBILESoft '19, p 99–110

Chen X, Zhao Y, Wang Q, Yuan Z (2018) MULTI: Multi-objective effort-aware just-in-time software defect prediction. Information and Software Technology 93:1–13, DOI 10.1016/j.infsof.2017.08.004, URL https://linkinghub.elsevier.com/retrieve/pii/S0950584917304627

Cohen P, West S, Aiken L (2014) Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences. DOI 10.4324/9781410606266

da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2017) A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering 43(7):641–657, DOI 10.1109/TSE.2016.2616306

Fan Y, Xia X, Alencar da Costa D, Lo D, Hassan AE, Li S (2019) The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction. IEEE Transactions on Software Engineering DOI 10.1109/TSE.2019.2929761

Farrar DE, Glauber RR (1967) Multicollinearity in regression analysis: The problem revisited. The Review of Economics and Statistics 49(1):92–107, URL http://www.jstor.org/stable/1937887

Fu W, Menzies T (2017) Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, Paderborn Germany, pp 72–83, DOI 10.1145/3106237.3106257, URL https://dl.acm.org/doi/10.1145/3106237.3106257

Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N (2014) An Empirical Study of Just-in-Time Defect Prediction Using Cross-Project Models. In: Proceedings of the 11th Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR 2014, pp 172–181, DOI 10.1145/2597073.2597075, URL https://doi.org/10.1145/2597073.2597075

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Transactions on Software Engineering 26(7):653–661, DOI 10.1109/32.859533

Guo PJ, Zimmermann T, Nagappan N, Murphy B (2010) Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, Association for Computing Machinery, New York, NY, USA, ICSE '10, p 495–504, DOI 10.1145/1806799.1806871, URL https://doi.org/10.1145/1806799.1806871

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings - International Conference on Software Engineering, pp 78–88, DOI 10.1109/ICSE.2009.5070510

Herzig K, Zeller A (2013) The impact of tangled code changes. In: 2013 10th Working Conference on Mining Software Repositories (MSR), pp 121–130, DOI 10.1109/MSR.2013.6624018

Hindle A, German DM, Holt R (2008) What do large commits tell us? a taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '08, p 99–108, DOI 10.1145/1370750.1370773, URL https://doi-org-s.nudtproxy.yitlink.com/10.1145/1370750.1370773

Ho TK (1995) Random Decision Forests. In: Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1, IEEE Computer Society, USA, ICDAR '95, p 278

Hoang T, Khanh Dam H, Kamei Y, Lo D, Ubayashi N (2019) DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In: IEEE International Working Conference on Mining Software Repositories, IEEE, vol 2019-May, pp 34–45, DOI 10.1109/MSR.2019.00016

Hoang T, Kang HJ, Lo D, Lawall J (2020) CC2Vec: distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ACM, Seoul South Korea, pp 518–529, DOI 10.1145/3377811.3380361, URL https://dl.acm.org/doi/10.1145/3377811.3380361

Huang Q, Xia X, Lo D (2017) Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, pp 159–170, DOI 10.1109/icsme.2017.51

Järvelin K, Kekäläinen J (2002) Cumulated gain-based evaluation of ir techniques. ACM Trans Inf Syst 20(4):422–446, DOI 10.1145/582415.582418, URL https://doi.org/10.1145/582415.582418

Jiang T, Tan L, Kim S (2013) Personalized defect prediction. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings, pp 279–289, DOI 10.1109/ASE.2013.6693087

Jiarpakdee J, Tantithamthavorn C, Treude C (2018) AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, Madrid, pp 92–103, DOI 10.1109/ICSME.2018.00018, URL https://ieeexplore.ieee.org/document/8530020/

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39(6):757–773, DOI 10.1109/TSE.2012.70

Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering 21(5):2072–2106, DOI 10.1007/s10664-015-9400-x, URL http://dx.doi.org/10.1007/s10664-015-9400-x

Kawata K, Amasaki S, Yokogawa T (2015) Improving relevancy filter methods for cross-project defect prediction. In: 2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence, pp 2–7, DOI 10.1109/ACIT-CSI.2015.104

Kim S, Zimmermann T, Pan K, Jr Whitehead EJ (2006) Automatic Identification of Bug-Introducing Changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp 81–90, DOI 10.1109/ASE.2006.23

Kim S, Whitehead EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34(2):181–196, DOI 10.1109/TSE.2007.70773

Kock N, Lynn G (2012) Lateral collinearity and misleading results in variance-based sem: An illustration and recommendations. Journal of the Association of Information Systems 13, DOI 10.17705/1jais.00302

Kondo M, German DM, Mizuno O, Choi EH (2020) The impact of context metrics on just-in-time defect prediction. Empirical Software Engineering 25(1):890–939, DOI 10.1007/s10664-019-09736-3

Koru AG, Zhang D, El Emam K, Liu H (2009) An investigation into the functional form of the size-defect relationship for software modules. IEEE Trans Softw Eng 35(2):293–304, DOI 10.1109/TSE.2008.90, URL https://doi.org/10.1109/TSE.2008.90

Krishna R, Menzies T, Fu W (2016) Too much automation? the bellwether effect and its implications for transfer learning. ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering pp 122–131, DOI 10.1145/2970276.2970339

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering 34(4):485–496, DOI 10.1109/TSE.2008.35

Lewis C, Lin Z, Sadowski C, Zhu X, Ou R, Whitehead EJ (2013) Does bug prediction support human developers? Findings from a Google case study. Proceedings - International Conference on Software Engineering pp 372–381, DOI 10.1109/ICSE.2013.6606583

Leys C, Ley C, Klein O, Bernard P, Licata L (2013) Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. Journal of Experimental Social Psychology 49(4):764–766, DOI https://doi.org/10.1016/j.jesp.2013.03.013, URL https://www.sciencedirect.com/science/article/pii/S0022103113000668

Li W, Zhang W, Jia X, Huang Z (2020) Effort-Aware semi-Supervised just-in-Time defect prediction. Information and Software Technology 126:106364, DOI 10.1016/j.infsof.2020.106364, URL https://linkinghub.elsevier.com/retrieve/pii/S0950584920301324

Lin D, Tantithamthavorn C, Hassan AE (2021) The Impact of Data Merging on the Interpretation of Cross-Project Just-In-Time Defect Models. IEEE Transactions on Software Engineering pp 1–1, DOI 10.1109/TSE.2021.3073920

Liu J, Zhou Y, Yang Y, Lu H, Xu B (2017) Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction. In: International Symposium on Empirical Software Engineering and Measurement, vol 2017-Novem, pp 11–19, DOI 10.1109/ESEM.2017.8

Ma Y, Luo G, Zeng X, Chen A (2012) Transfer learning for cross-company software defect prediction. Information and Software Technology 54(3):248–256, DOI 10.1016/j.infsof.2011.09.007, URL http://dx.doi.org/10.1016/j.infsof.2011.09.007

Matsumoto S, Kamei Y, Monden A, Matsumoto Ki, Nakamura M (2010) An analysis of developer metrics for fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, Association for Computing Machinery, New York, NY, USA, PROMISE '10, DOI

10.1145/1868328.1868356, URL https://doi.org/10.1145/1868328.1868356

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering - PROMISE '09, ACM Press, Vancouver, British Columbia, Canada, p 1, DOI 10.1145/1540438.1540448, URL http://portal.acm.org/citation.cfm?doid=1540438.1540448

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: Current results, limitations, new approaches. Automated Software Engineering 17(4):375–407, DOI 10.1007/s10515-010-0069-5

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Technical Journal 5(2):169–180, DOI 10.1002/bltj.2229

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings - International Conference on Software Engineering, vol 2005, pp 284–292, DOI 10.1109/ICSE.2005.1553571

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '06, p 452–461, DOI 10.1145/1134285.1134349, URL https://doi-org-s.nudtproxy.yitlink.com/10.1145/1134285.1134349

Nakagawa S, Schielzeth H (2013) A general and simple method for obtaining R2 from generalized linear mixed-effects models. Methods in Ecology and Evolution 4(2):133–142, DOI 10.1111/j.2041-210x.2012.00261.x

Nam J, Pan SJ, Kim S (2013) Transfer defect learning. In: 2013 35th International Conference on Software Engineering (ICSE), pp 382–391, DOI 10.1109/ICSE.2013.6606584

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering 31(4):340–355, DOI 10.1109/TSE.2005.49

Pan SJ, Yang Q (2010) A survey on transfer learning. IEEE Transactions on Knowledge and Data Engineering 22(10):1345–1359, DOI 10.1109/TKDE.2009.191

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12:2825–2830

Pornprasit C, Tantithamthavorn CK (2021) JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, Madrid, Spain, pp 369–379, DOI 10.1109/MSR52588.2021.00049, URL https://ieeexplore.ieee.org/document/9463103/

Purushothaman R, Perry D (2005) Toward understanding the rhetoric of small source code changes. IEEE Transactions on Software Engineering 31(6):511–526, DOI 10.1109/TSE.2005.74

Rahman F, Posnett D, Devanbu P (2012) Recalling the "imprecision" of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12, ACM Press, Cary, North Carolina, p 1, DOI 10.1145/2393596.2393669, URL http://dl.acm.org/citation.cfm?doid=2393596.2393669

Shihab E, Hassan AE, Adams B, Jiang ZM (2012) An industrial study on the risk of software changes. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012 DOI 10.1145/2393596.2393670

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, pp 1–5, DOI 10.1145/1083142.1083147

Spadini D, Aniche M, Bacchelli A (2018) PyDriller: Python framework for mining software repositories. In: ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 908–911, DOI 10.1145/3236024.3264598

Tabassum S, Minku LL, Feng D, Cabral GG, Song L (2020) An investigation of cross-project learning in online just-in-time so ware defect prediction. In: Proceedings - International Conference on Software Engineering, pp 554–565, DOI 10.1145/3377811.3380403

Tan M, Tan L, Dara S, Mayeux C (2015) Online Defect Prediction for Imbalanced Data. In: Proceedings - International Conference on Software Engineering, vol 2, pp 99–108, DOI 10.1109/ICSE.2015.139

Tantithamthavorn C, Hassan AE (2018) An experience report on defect modelling in practice: Pitfalls and challenges. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp 286–295

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2017) An empirical comparison of model validation techniques for defect prediction models. IEEE Transactions on Software Engineering 43(1)

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2018) The impact of automated parameter optimization for defect prediction models. IEEE Transactions on Software Engineering

Tosun A, Bener A (2009) Reducing false alarms in software defect prediction by decision threshold optimization. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, USA, ESEM '09, p 477–480

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14(5):540–578, DOI 10.1007/s10664-008-9103-7

Wang S, Liu T, Nam J, Tan L (2020) Deep Semantic Feature Learning for Software Defect Prediction. IEEE Transactions on Software Engineering 46(12):1267–1293, DOI 10.1109/TSE.2018.2877612

Wu R, Zhang H, Kim S, Cheung SC (2011) ReLink: Recovering Links between Bugs and Changes. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE '11, pp 15–25, DOI 10.1145/2025113.2025120, URL https://doi.org/10.1145/2025113.2025120

Yan M, Xia X, Fan Y, Hassan AE, Lo D, Li S (2020) Just-In-Time Defect Identification and Localization: A Two-Phase Framework. IEEE Transactions on Software Engineering PP(c):1, DOI 10.1109/TSE.2020.2978819

Yang X, Lo D, Xia X, Zhang Y, Sun J (2015) Deep Learning for Just-in-Time Defect Prediction. In: Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, 1, pp 17–26, DOI 10.1109/QRS.2015.14

Yang X, Lo D, Xia X, Sun J (2017) TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology 87:206–220, DOI 10.1016/j.infsof.2017.03.007, URL http://dx.doi.org/10.1016/j.infsof.2017.03.007

Yang Y, Zhou Y, Liu J, Zhao Y, Lu H, Xu L, Xu B, Leung H (2016) Effort-Aware just-in-Time defect prediction: Simple unsupervised models could be better than supervised models. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, vol 13-18-Nove, pp 157–168, DOI 10.1145/2950290.295035

Zeng Z, Zhang Y, Zhang H, Zhang L (2021) Deep Just-in-Time Defect Prediction: How Far Are We? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2021, pp 427–438, DOI 10.1145/3460319.3464819, URL https://doi.org/10.1145/3460319.3464819, event-place: Virtual, Denmark

Zhang F, Mockus A, Zou Y, Khomh F, Hassan AE (2013) How Does Context Affect the Distribution of Software Maintainability Metrics? In: 2013 IEEE International Conference on Software Maintenance, IEEE, Eindhoven, Netherlands, pp 350–359, DOI 10.1109/ICSM.2013.46, URL http://ieeexplore.ieee.org/document/6676906/

Zhang F, Mockus A, Keivanloo I, Zou Y (2014) Towards building a universal defect prediction model. In: Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014, ACM Press, Hyderabad, India, pp 182–191, DOI 10.1145/2597073.2597078, URL http://dl.acm.org/citation.cfm?doid=2597073.2597078

Zhou ZH (2012) Ensemble methods: Foundations and algorithms

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE '09, pp 91–100, DOI 10.1145/1595696.1595713, URL https://doi.org/10.1145/1595696.1595713