JIT-Align: A Semantic Alignment-Based Ranking Framework for Just-In-Time Defect Prediction

Yujie Ye[1], Huiqun Yu[*1,2], Guisheng Fan[*1,2], Yuguo Liang[1], Jianan Dong[1], Wentao Chen[1]
[1]Department of Computer Science and Engineering,
East China University of Science and Technology, Shanghai 200237, China
[2]Shanghai Engineering Research Center of Smart Energy, Shanghai 201103, China

***Abstract*—To promptly identify software defects and prevent defective code changes from being integrated into the repository, Just-In-Time Software Defect Prediction (JIT-SDP) has demonstrated promising research findings. Recent studies have begun to utilize Pre-trained Models (PTMs) for training and prediction, yet these models inherently impose input length limitations, leading to forced truncation of inputs. However, previous work has largely overlooked the impact of forced truncation, even though it may inadvertently discard critical input information, leading to degraded model performance. Moreover, some existing methods fail to maintain consistency in truncation during each model construction process, leading to unexplainable truncations and unstable model performance. In addition, previous datasets suffer from limitations and incompleteness. To this end, we construct a large-scale and comprehensive dataset, MC4Defect. Moreover, we propose JIT-Align, which prioritizes code changes within a commit using a semantic alignment algorithm to make full use of the limited input space of PTMs. To evaluate the feasibility of JIT-Align, we first assess the classification capability of our method by comparing it against four baselines across five datasets. Then, we conduct ablation studies on the proposed semantic alignment framework to validate its effectiveness. Experimental results show that JIT-Align, along with its semantic alignment framework, outperforms all baselines in JIT-SDP tasks, with average F1 score improvements of 3.1%-9.6% and MCC increases of 3.1%-9.7% across all projects, exhibiting higher stability and better interpretability compared to alternative approaches.**

***Index Terms*—Just-In-Time, Defect Prediction, Pre-trained Model, Deep Learning**

I. INTRODUCTION

In the process of software development, it is inevitable for developers to introduce defects into the code repository through code commits. If these defects are overlooked by quality assurance personnel and deployed to production environments, they can lead to poor user experiences and significant economic losses. To address this challenge, researchers have explored automated defect detection at various granularities of software components, such as software methods [1]–[3], modules [4]–[6] and file-level [7] defect prediction. These approaches aim to utilize limited resources for defect inspection and repair, thereby enhancing the efficiency of software quality assurance (QA). However, modern software development is collaborative and agile, with a prevailing trend towards continuous delivery and deployment. The goal is to build, modify, and release software at a faster pace and with greater frequency. Under this growing trend, developers are more likely to introduce defects into software, and the task of detecting and fixing these defects becomes increasingly difficult and costly. To tackle this challenge, researchers have proposed a new subfield of software defect prediction (SDP) known as JIT-SDP. The objective of JIT-SDP is to predict the likelihood of defects in each software commit, allowing for immediate alerts before defective code is merged into the repository, thereby improving the quality of the repository's code.

Early studies constructed JIT-SDP prediction models based on machine learning or deep learning techniques. In recent years, with the advancement and achievements of large-scale pre-trained language models (PLMs), some researchers have begun to explore their applications in defect prediction. Ni et al. [8] were the first to employ CodeBERT [9] for JIT-SDP, achieving promising results. Additionally, several other pre-trained models (PTMs), such as CodeT5 [10] and CCT5 [11], have been applied to JIT-SDP tasks. However, PTMs enforce strict input length limitations (e.g., 512), while the required input length for semantic information varies across different commits. This inevitably leads to forced truncation, which may cause critical information to be discarded while allowing irrelevant information to be retained as noise. Such issues can degrade model performance and introduce uncertainty. Despite this, little prior work has addressed or attempted to mitigate the impact of forced truncation in PTMs for JIT-SDP. Existing approaches often fail to distinguish between critical and non-critical input information for the limited input length. Some methods even truncate different, random content across training runs, resulting in significant variations in model performance, which severely impacts the model's interpretability and stability. Beyond truncation issues, commonly used JIT-SDP datasets also exhibit certain limitations, such as being outdated, poor scalability, and incomplete information. These constraints may further impact the completeness and reliability of JIT-SDP.

To address the aforementioned limitations, we introduce a new, comprehensive dataset for JIT-SDP, named **MC4Defect**, which stands for 'A More Complete Dataset for JIT-SDP'. We

*Corresponding Authors: Huiqun Yu (yhq@ecust.edu.cn), Guisheng Fan (gsfan@ecust.edu.cn)

compare the impact of different truncation content on classification performance and stability across two different pre-trained models. Furthermore, we introduce a novel approach, **JIT-Align**, which integrates the semantic-align framework to reduce the truncation of key information, thereby fully utilizing the limited input space of the PTM. To the best of our knowledge, JIT-Align is the first method to consider PTM truncation in the JIT-SDP field, focusing on enhancing model performance and stability. Specifically, JIT-Align consists of the following key steps: (1) Extract expert features and initial semantic representations from commits. (2) Build a FAISS[1] database using PTMs to generate vector representations of commit messages and file-level code changes. (3) Compute similarity scores between each file's code changes and its commit message using vector similarity algorithms and storing the ranking results. (4) Prioritize the ranked file-level code changes when feeding them into the model, ensuring that the model learns the most critical semantic information first. (5) Integrate the prediction results of the expert feature-based model through model fusion to predict defects in a commit. Eventually, this paper makes the main contributions as below:

- **Dataset MC4Defect**. We construct a large-scale dataset based on nearly a decade (2015–2024) of commits from five open repositories, containing 258,243 commits. This dataset provides a complete diff context, including all committed files without limiting the number of files or modified lines, and offers file-level defect labels.
- **Analysis of truncation effects on models**. We analyze the performance differences of models across five datasets when truncation content is not fixed. The results reveal that varying truncation content during model training causes significant fluctuations in model performance, (with variations of over 13% in F1-score and more than 7% in MCC for CodeT5).
- **JIT-Align**. We propose JIT-Align, a semantic alignment-based JIT-SDP approach. Considering the forced truncation issue in PTMs, JIT-Align leverages the matching ranking between commit messages and code changes to effectively utilize the fixed input length of PTMs, ensuring model performance and stability. To facilitate future research, we publicly release the replication package for JIT-Align along with the MC4Defect dataset [2]. Due to space constraints, this address also provides results on similarity algorithms and parameter tuning for the semantic alignment module.

The remainder of this paper is organized as follows. Section II presents the construction and details of the proposed dataset. Section III describes the design of JIT-Align. Following that, Section IV outlines the experimental setup, and Section V reports the experimental results. Section VI introduces the related work. Section VII discusses potential threats to validity. Finally, Section VIII concludes our work and outlines future research directions.

[1]https://github.com/facebookresearch/faiss

[2]https://github.com/xiqiaong/JIT-Align.git

II. DATASET CONSTRUCTION

A. Motivation

The current SOTA JIT-SDP methods mainly rely on two datasets: (1) Research based on the Defect4J [8], which is an extension of the LLTC4J [12] focusing only on Java repositories. The studies based on this dataset include JITFine [8], JIT-Smart [13]; (2) Research based on the LAPredict [14] dataset, which collects data from four programming languages and six influential open-source repositories. The studies based on this dataset include LAPredict [14] and SimCom [15].

Unfortunately, these mainstream datasets have some limitations. **First, these datasets contain early samples.** For example, the latest commit in the Defect4J comes from 2019, with the oldest data dating back to 2001. The LAPredict dataset collects historical commits from open-source repositories between 2011 and 2020, with most of the latest commits from 2019 (due to filtering rules [16]). As time progresses, modern software development practices and frameworks evolve, and the best-performing model on older datasets may not perform well in recent development environments. Validating the model on outdated datasets is no longer meaningful. **Second, these datasets have structural and semantic limitations.** For example, the Defect4J uses a set data structure to store the added and removed code lines for each file in a commit. Due to the characteristics of the set data structure, when reading the dataset during model training, even with a fixed random seed, the code changes are different each time. Based on the previous description of PTM truncation, it can be inferred that the data retained for training during each training runs is different, which leads to unstable evaluation results for the model each time. Additionally, shuffling the added and removed lines for files may break the contextual semantics of sequential lines with dependencies. For the LAPredict dataset, it only retains the first ten file changes from the original commit and the first ten lines of each changed file, making the data incomplete. Furthermore, this dataset is not conducive to extracting code structure information.

To address the limitations of the above datasets and improve their completeness and scalability, this paper constructs a new dataset: MC4Defect.

TABLE I
STATISTICS OF THE STUDIED DATASET: MC4DEFECT

Project	#Changes	%Defect	Language	#Repos
Gerrit	36,777	13.44%	Java	12
JDT	7,385	44.25%	Java	3
Openstack	48,790	26.84%	Python	23
Platform	23,705	28.89%	Java	15
QT	141,586	22.30%	C++	23

B. MC4Defect Construction

1) Project and repository selection: To ensure the quality and diversity of the data, we consider popular open-source projects and core development repositories, covering the main

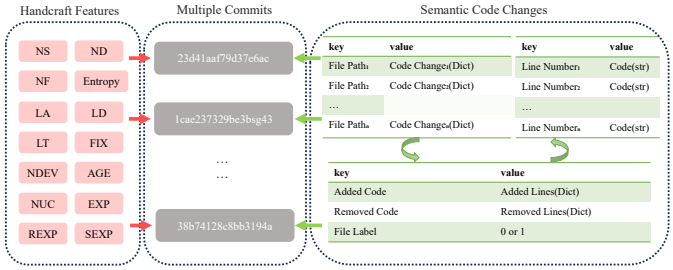

Fig. 1. Design of the MC4Defect Data Structure.

programming languages: C++, Python, and Java. These open-source projects and repositories all have standardized development and review processes, as shown in Table I (Due to the variation in issue management between migrated project repositories, we select older repository management formats for consistency).

2) Dataset construction: Specifically, for each project, we collect all code commits between 2015-01-01 and 2024-12-01. Then, based on prior research [17]–[19], we apply the widely used SZZ algorithm [20], [21] to identify defect-fixing commits. Initially, we analyze commit messages to detect defect fixes. Then, we utilize Git diff to pinpoint the modified lines, indicating the lines responsible for introducing defects. Next, we employ Git blame to identify the specific commit that introduced the defect, classifying it as a defect-introducing commit. Commits not marked as defect-related by the SZZ algorithm were categorized as clean commits. We then filter the labeled commits based on established methods [16], removing irrelevant elements such as code comments, empty changes, suspicious fixes, and suspicious introducing commits. Additionally, we consider the defect introduction rate trends of each project. We assume that high-quality datasets are derived from stable development phases, with a sharp increase in defect introduction indicating an unstable development cycle or authentication latency [22]. Subsequently, we design the data structure for the new dataset (as shown in Figure 1), ensuring that it is complete and scalable. In addition, we create the expert feature dataset based on the features proposed by Kamei et al. [16]. Ultimately, we obtain 258,243 real commits across these 5 projects (as shown in Table I).

III. APPROACH

Figure 2 shows the basic implementation architecture of our method, JIT-Align, which primarily consists of 4 parts: data processing, vector generation & storage, vector semantic alignment & classification, and output.

A. Data Preprocessing

First, the historical commit data of the project is obtained from the Git repository and processed (detailed in Section II). Each commit mainly includes the commit message and the code changes of various files.

B. Vector Generation and Storage

At this stage, to semantically compare the matching degree of numerous file code changes with the commit intent (commit message), we use the CodeT5 encoding module to generate 768-dimensional vector representations containing semantic context for both the commit message and the code changes of each file. To facilitate reuse and reduce subsequent computational overhead, we store these vectors in a FAISS vector database.

1) Vector generation: **Tokenization**. CodeT5 uses the PLBart [23] tokenizer, which first splits text or code sequences into subwords: $T = [\text{[CLS]}, T_1, T_2, \ldots, T_n, \text{[SEP]}]$. Then, each token is assigned a unique ID from the vocabulary ($ID = [101, ID_1, ID_2, \ldots, 102]$). **Embedding**. The token IDs are transformed into high-dimensional vectors through the embedding layer for processing by the Transformer. The word embedding matrix W_e is first generated, with the shape of:

$$W_e \in \mathbb{R}^{V \times d} \tag{1}$$

Here, V represents the vocabulary size, and $d = 768$ denotes the embedding dimension. For the Token IDs, the embedding layer retrieves the corresponding row in W_e as follows:

$$E = \begin{bmatrix} W_e[101] \\ W_e[\text{ID}_1] \\ W_e[\text{ID}_2] \\ \vdots \\ W_e[102] \end{bmatrix} \in \mathbb{R}^{L \times d} \tag{2}$$

Here $d = 768$ denotes the embedding dimension. Here L represents the sentence length, and E denotes an $L \times 768$-dimensional matrix, where each row corresponds to a 768-dimensional vector. **Encoder & Pooling**. For each word embedding matrix, the multi-layer attention mechanism extracts contextual information, allowing each token to contain contextual information. Then, through residual connections, layer normalization, and feed-forward networks, after N layers of Transformer, the $L \times 768$ dimensional token representations are aggregated into a single 768-dimensional vector. In this work, average pooling is used for this module:

$$V = \frac{1}{L} \sum_{i=1}^{L} H_i \tag{3}$$

Where $H_i \in \mathbb{R}^{768}$ is the vector of the i-th token computed by Transformer, and $V \in \mathbb{R}^{768}$ is finally stored in the FAISS database.

2) FAISS storage: FAISS vector database is specifically designed for large-scale vector indexing and nearest neighbor search, and it can be accelerated using GPUs. In this paper, we use `IndexFlatL2` to store 768-dimensional vectors. To map the vectors, we utilize the commit hash and the file path names in the commit. Specifically:

$$\text{commit}_{id} \rightarrow \text{msg}_{index} \tag{4}$$

$$(\text{commit}_{id}, \text{file}_{path}) \rightarrow \text{file}_{index} \tag{5}$$

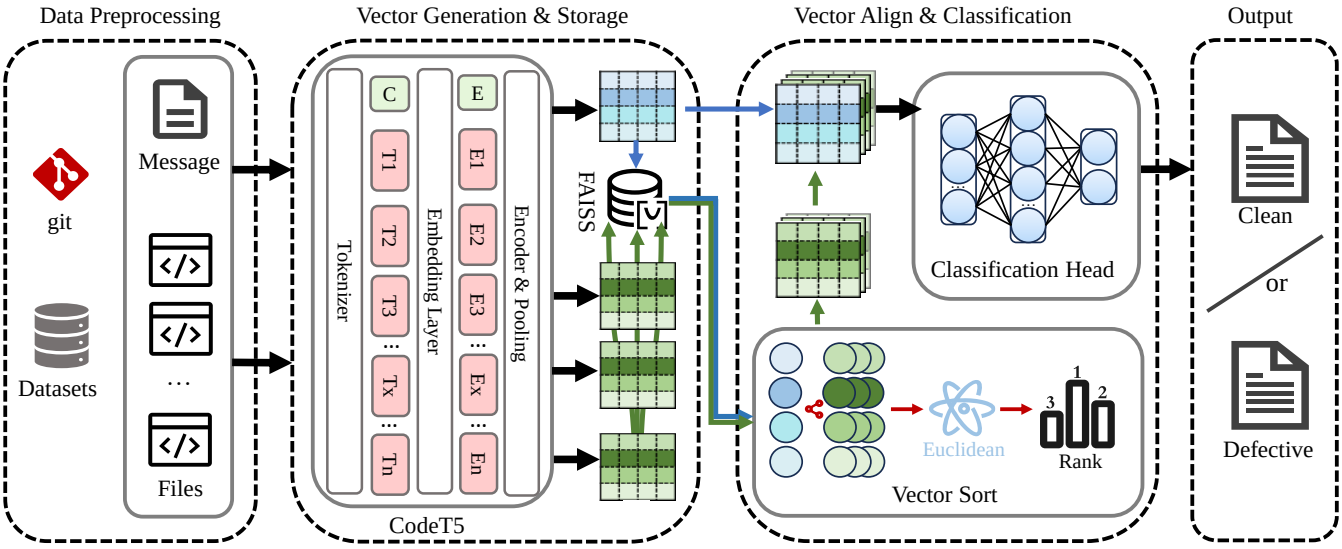

Fig. 2. Framework of JIT-Align.

The process primarily generates three main files: msg_faiss.index, file_faiss.index, and the mapping file faiss_mappings.pkl containing msg_map, file_map, msg_keys, and file_keys.

C. Vector Semantic Alignment and Classification Model

1) Semantic alignment: To obtain the file code changes that most closely match the commit message, based on the Git-recommended sorting, we use Euclidean distance to calculate the similarity between each file's code change and the commit message vector. This reduces the possibility of important information being truncated by the model and increases the weight of important information in the sequence, thus improving model performance and truncation interpretability. The specific implementation is shown in Equation 6 and Algorithm 1.

$$d(A,B) = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2} \tag{6}$$

Where A and B are two vectors (e.g., 768-dimensional vectors generated by CodeT5). A_i and B_i are the values of A and B at the i-th dimension, respectively.

2) Classification: The sorted features are input into the classification head for classification and mapped to the final classification score. After passing through the hidden layer, extraction layer, dropout layer, and linear transformation, the output logits are passed into the loss function for sigmoid activation.

IV. EXPERIMENTAL SETTINGS

To verify the impact of model truncation on JIT-SDP and the effectiveness of the proposed JIT-Align method, this paper raises the following research questions for experiment and discussion, as detailed below:

Algorithm 1: Sorting Files by Vector Similarity (Semantic Alignment)

Input: commit hash *H = hash_id*
code changes of files *F = [f1, f2, ..., fn]*
Output: sorted code changes of files *F_Sorted = [f1', f2', ..., fn']*
align scores *S = [s1, s2, ..., sn]*

(1) load indices: *msg_index* <- msg_faiss.index; *file_index* <- file_faiss.index
(2) load mapping file: *mappings* <- faiss_mappings.pkl
(3) *msg_map* <- *mappings*["msg_map"]; *file_map* <- *mappings*["file_map"];
(4) use msg_key *H* to look up the index in *msg_map*: *msg_map*[*H*]
(5) use *msg_map*[*H*] to extract the commit message vector: *msg_vector*
(6) retrieve the original file order: *file_order*
(7) initialize the score list *S*
(8) **for each** file path *f_path* in *F*.keys(), **do**
(9) compute the *file key*: *file_key* = (*H*, *f_path*)
(10) use *file_key* to retrieve the index from *file_map*: *file_map*[*file_key*]
(11) use *file_map*[*file_key*] to extract the file change vector: *file_vector*
(12) *similarity* = calculate_similarity(*msg_vector*, *file_vector*)
(13) compute and normalize the path score using *file_order*: *path_score*
(14) *final_score = alpha * path_score + beta * similarity*
(15) append (*f_path*, *final_score*) to the list *S*
(16) **end for**
(17) *S*.sort(*key*=lambda *x*: *x*[1], *reverse*=*True*)
(18) return *F_Sorted* = {*file_name*: *F*[*file_name*] for *file_name*, s in *S*}

- **RQ1: Does the variation in truncated content affect the performance of JIT-SDP?**
- **RQ2: Can the proposed JIT-Align achieve better performance in JIT-SDP?**
- **RQ3: Is the vector-align module truly effective?**

A. Datasets

In section II, we collect all commits from 76 projects across five popular and well-managed open-source projects over the past 10 years, and perform the corresponding data preprocessing, resulting in a total of 258,243 commits records. To maximize the effectiveness of methods, and considering concept drift [24] and stable development cycles, we observe

the defect rate across the entire commit timeline for all projects, as shown in Figure 3. Upon examining the Figure 3, we select different experimental data intervals from the filtered dataset for each project and split the dataset according to the recommended ratio of 0.75: 0.05: 0.2 [14]. The final results are shown in Table II. The "%Trunc." column shows the proportion of commits truncated by the PTMs in each dataset.

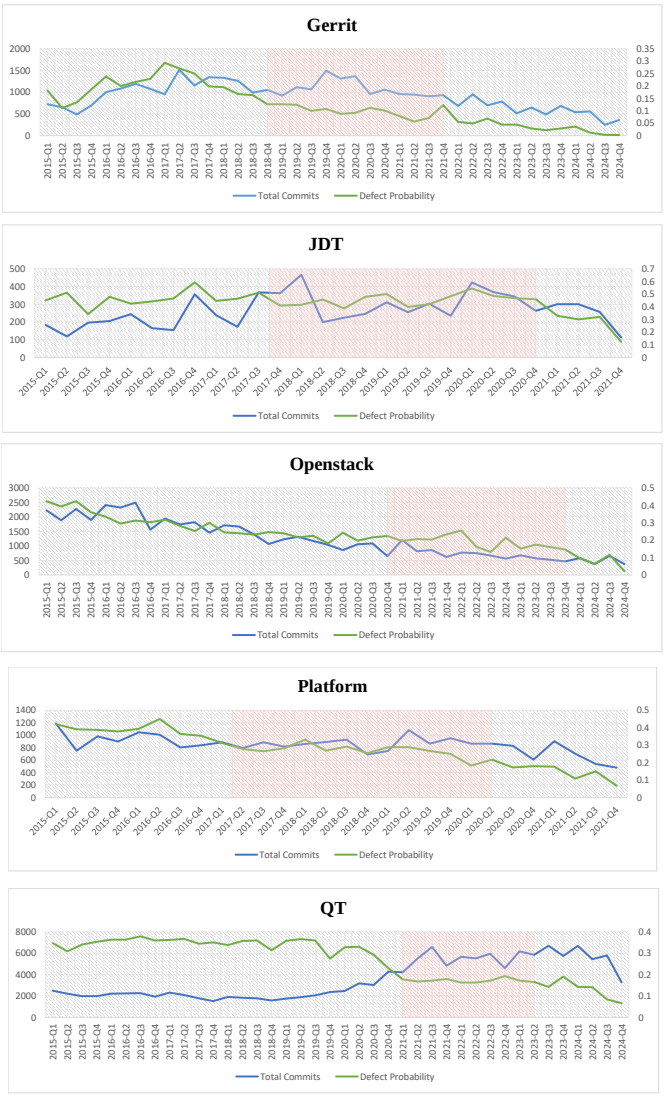

Fig. 3. Variation and selection of the project's defect rate.

B. Baselines

To validate the effectiveness of JIT-Align, this paper selects recent SOTA models and methods as follows:

- **LApredict [14]:** This method aims to utilize traditional expert feature information, such as "number of added lines of code", and constructs a defect prediction model using a logistic regression classifier.
- **JITFine [8]:** This is a hybrid method that combines semantic semantic features (SF) extracted from syntactic structures, along with expert features (EF).
- **JIT-Smart [13]:** JIT-Smart proposed a multi-task learning framework for instant defect prediction and localization. It can both identify commits that introduce defects and locate the lines that introduce the defects.
- **CCT5 [11]:** A PTM specifically designed for code changes. These models leverage domain knowledge hidden in a large amount of unlabeled data and ensure that the learned knowledge is fully utilized during fine-tuning for code change-related tasks.

TABLE II
DATASETS FROM MC4DEFECT

Project	#Commits	%Defect (train-valid-test)	%Trunc.
Gerrit	14,105	10.67%-8.09%-8.47%	34.53%
JDT	4,004	45.22%-47%-47.69%	53.70%
Openstack	12,651	21.50%-19.46%-22.52%	50.96%
Platform	10,959	28.20%-25.23%-21.11%	31.02%
QT	54,726	17.16%-17.51%-17.14%	27.22%
Total	96,445	-	32.94%

C. Evaluation Metrics

To evaluate the effectiveness of baselines and JIT-Align, we select the following evaluation metrics:

AUC-ROC [25] indicates the accuracy of the model under different threshold conditions, and it is particularly useful for comparing the performance of different models, especially in imbalanced datasets. **F1-score** [26] is a popular metric used in many previous software engineering studies, and it is defined as the harmonic mean of precision and recall: $F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$. For a commit in the test dataset, there are four possible prediction outcomes: a commit that contains an actual defect is predicted as defective (True Positive, TP); a commit that is actually clean is predicted as defective (False Positive, FP); a commit that is actually defective is predicted as clean (False Negative, FN); and a commit that is actually clean is predicted as clean (True Negative, TN). Based on these 4 possible outcomes, **precision** is the ratio of correctly predicted defective commits to all commits predicted as defective: Precision $= \frac{TP}{TP+FP}$. **Recall** is the ratio of correctly predicted defective commits to all actual defective commits: Recall $= \frac{TP}{TP+FN}$. **Matthews Correlation Coefficient (MCC)** [27] is a comprehensive metric that considers TP, TN, FP, and FN, and is considered a very effective performance evaluation metric for imbalanced datasets [28]. The formula is: MCC $= \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$.

D. Experiment Environment

The experiments in this paper are conducted on an RTX 3090 (24GB), with a 14 vCPU Intel(R) Xeon(R) Platinum 8362 CPU @ 2.80GHz. Additionally, the operating system used for dataset construction is Windows 11 Professional 64-bit. To recreate this dataset, ensure that the available disk space exceeds 50GB and the memory capacity is at least 32GB.

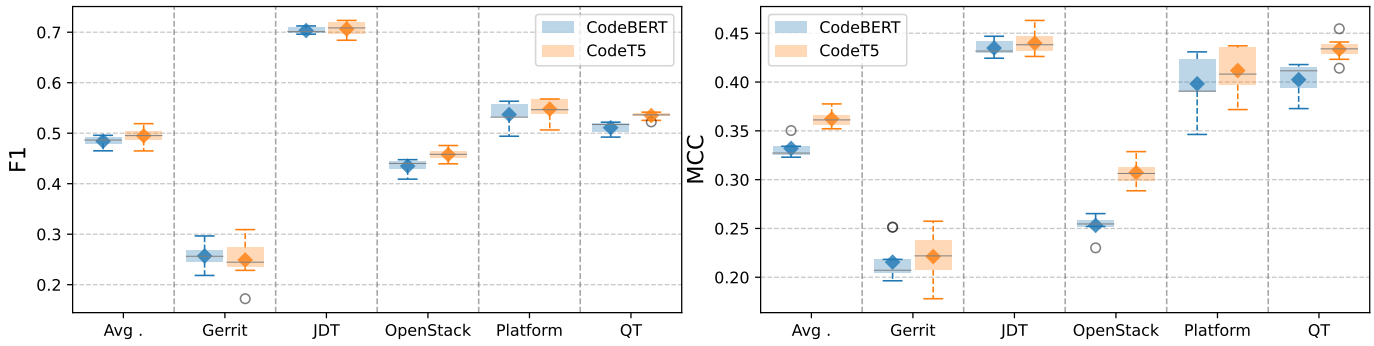

Fig. 4. Performance fluctuations of different PTMs under different input sequences.

V. EXPERIMENTAL RESULTS

A. RQ1: Does the variation in truncated content affect the performance of JIT-SDP?

As shown in Table II, more than 30% of the commits in almost all projects suffer from truncation issues when fed into the pre-trained model. The truncation problem is particularly prevalent in the JDT project dataset (53.70%), while it is relatively less severe in the Qt project dataset (27.22%). These truncation issues may cause the model to overlook important information during training, thereby posing a threat to its performance and stability.

To further investigate the impact of truncation on model performance, in this research question, we use CodeBert [9] and CodeT5 [10] as representative PTMs. The former is a commonly used PTM in past JIT-SDP tasks [8], [13], while the latter is one of the top-performing PTMs in recent years. We simulate random truncation scenarios and conduct 10 experiments on five datasets. In each experiment, the code changes for training and validation vary, while the test set remains constant.

The experimental results are shown in Figure 4. Regardless of whether CodeBERT or CodeT5 is used, variations in code change order and content lead to significant performance fluctuations. For example, Gerrit exhibits the most noticeable fluctuation in F1-score, with a variation exceeding 13.69% for CodeT5-based models. On the other hand, Qt shows the smallest fluctuation when using the CodeT5 pre-trained model, with a variation of at least 1.93%, while JDT has the smallest fluctuation for CodeBERT, with a range of 1.63%. Similarly, MCC also exhibits considerable variation under different truncation conditions. When using CodeT5, Gerrit remains the most unstable project, with fluctuations exceeding 7.94%. JDT has the smallest variation, with a performance gap of 3.69% between the best and worst models. For CodeBERT, Platform shows the largest fluctuation at 8.47%, while JDT is the most stable, with a variation of 2.25%.

Based on the above observations, we can reasonably infer that different code changes within a commit have varying priorities in facilitating model learning. If relevant content is truncated while irrelevant content is learned, model performance may degrade.

Furthermore, CodeT5 outperforms CodeBERT in both metrics across most projects. On average, the F1-score of CodeT5-based models ranges from 46.49% to 51.89%, whereas CodeBERT-based models achieve only 46.51% to 49.58%. A similar trend is observed for MCC. In particular, the MCC of CodeT5-based models ranges from 35.21% to 37.76%, whereas CodeBERT-based models achieve only 32.30% to 35.03%.

> **Answer to RQ1**: In summary, regardless of whether CodeBERT or CodeT5 is used, forced truncation in pre-trained models significantly impacts performance, thereby affecting the stability of the model. Additionally, the model based on CodeT5 outperforms CodeBERT on both evaluation metrics across most projects.

B. RQ2: Can the proposed JIT-Align achieve better performance in JIT-SDP?

In this research question, we compare JIT-Align with the SOTA methods in the field of JIT-SDP to evaluate its effectiveness and stability. We follow the original settings of the baselines' articles. Since some baselines cannot ensure the consistency of the truncation content in each run, we repeat five experiments on five datasets for these baselines to ensure fairness.

The Table III presents the results of the baseline models and our proposed method, JIT-Align, on five datasets. As shown in the five tables, JIT-Align performs well across multiple metrics and datasets. Specifically, on the Gerrit dataset, JIT-Align outperforms the baselins in Precision, F1, and MCC metrics, with performance improvements of 14.03%, 5.56%, and 4.94% over the worst-performing models, respectively. Additionally, surprisingly, the LAPredict model achieves over 62% in Recall, indicating that it is able to detect most of the positive cases, although there is a high false positive rate. On the JDT dataset, JIT-Align also performs reasonably well. Except for Precision, it improves the other metrics by 28.54%, 12.03%, and 11.25%, respectively, compared to the worst-performing model. Among these, LAPredict performs best on the Precision metric, and this is notably different

TABLE III
PERFORMANCE OF DIFFERENT METHODS ON MULTIPLE DATASETS

Gerrit				
Methods	Recall	Precision	F1	MCC
LAPredict	**62.34**	21.01	31.43	26.10
JITFine	26.36	33.16	29.37	23.82
JIT-Smart	27.20	33.16	29.16	23.76
CCT5	33.05	32.78	32.92	26.68
JIT-Align	34.31	**35.04**	**34.72**	**28.70**
JDT				
Methods	Recall	Precision	F1	MCC
LAPredict	51.83	**75.86**	61.59	39.21
JITFine	78.74	68.28	73.10	45.57
JIT-Smart	64.19	70.29	67.03	39.76
CCT5	77.23	62.11	68.84	34.84
JIT-Align	**80.37**	67.92	**73.62**	**46.09**
Openstack				
Methods	Recall	Precision	F1	MCC
LAPredict	42.81	39.17	40.91	22.77
JITFine	38.95	41.42	40.14	23.45
JIT-Smart	38.95	**52.24**	44.62	31.96
CCT5	**65.26**	37.13	47.33	28.30
JIT-Align	60.00	42.12	**49.49**	**32.24**
Platform				
Methods	Recall	Precision	F1	MCC
LAPredict	56.59	48.16	52.04	38.07
JITFine	52.27	45.57	48.69	33.88
JIT-Smart	61.99	**50.71**	55.78	42.77
CCT5	**77.11**	39.40	52.15	37.60
JIT-Align	64.79	50.08	**56.50**	**43.52**
QT				
Methods	Recall	Precision	F1	MCC
LAPredict	37.26	33.40	35.22	20.98
JITFine	56.72	51.38	53.91	43.89
JIT-Smart	54.21	50.52	52.30	42.05
CCT5	**65.72**	43.45	52.31	41.30
JIT-Align	57.41	**52.56**	**54.88**	**45.11**
Average				
Methods	Recall	Precision	F1	MCC
LAPredict	50.17	43.52	44.24	29.43
JITFine	50.61	47.96	49.04	34.12
JIT-Smart	49.31	**51.38**	49.78	36.06
CCT5	**63.67**	42.97	50.71	33.74
JIT-Align	59.38	49.54	**53.84**	**39.13**

from its performance on the Gerrit dataset, indicating the risk and uncertainty of building models based on a single feature. For the Openstack dataset, JIT-Align improves the F1 and MCC metrics by 9.35% and 9.47%, respectively, over the worst baseline model, suggesting that JIT-Align has good classification ability. However, its performance in Recall and Precision is more modest. On the Platform dataset, JIT-Align improves the F1 and MCC metrics by 7.81% and 9.64%, respectively, over the worst baseline model. CCT5 performs best in Recall, showing that this model can detect a large proportion of defective commits in this dataset, although it comes with a higher false positive rate. Finally, for the QT dataset, JIT-Align achieves the best performance in Precision, F1, and MCC, with improvements of 19.16%, 19.66%, and 24.13%, respectively, while CCT5 performs best in Recall.

In addition, we expand the experimental results by calculating the average performance of each baseline method across the five datasets, in order to evaluate the performance of JIT-Align from a holistic perspective. The results show that our method outperforms all baseline methods in terms of classification ability (F1 improved by more than 9.6% compared to the worst-performing method, and MCC improved by more than 9.7% compared to the worst-performing method).

Answer to RQ2: Across all projects, the proposed JIT-Align method outperforms other baselines in classification performance. Compared to JIT-Fine and JIT-Smart, JIT-Align maintains consistency in truncated content across multiple training runs, improving the stability of model performance.

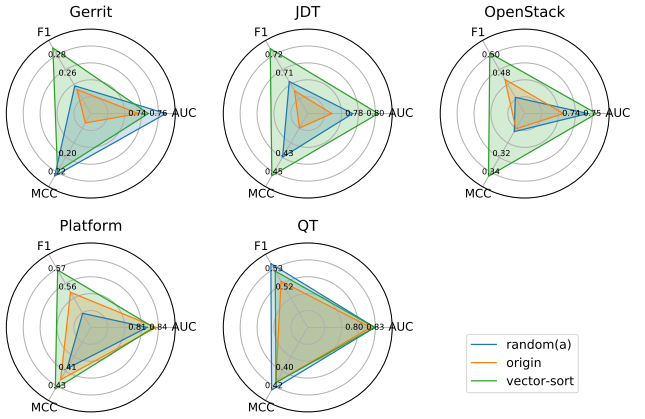

Fig. 5. Results under different input modes.

C. RQ3: Is the vector-align module truly effective?

To further investigate the impact of the vector-align module on model performance and stability, we arrange code changes at the file level for pre-trained model input using three optional modes: vector-sort, random(a), and origin. We then compare the model's performance under these different settings. Vector-sort refers to the vector-align mode, random(a) represents the random mode with results averaged over 10 runs, and origin follows the default Git-recommended order.

From Figure 5, it can be observed that the green section represents the algorithm module proposed in this paper, the blue section represents the average of the random mode, and the orange section represents the Git recommended default code change order. In terms of the three-dimensional metrics (AUC, F1, MCC), our proposed Semantic-Align module not only interpretably sorts important file code changes to the front of the input sequence, but also significantly improves model performance. In the QT dataset, there is little difference in

model performance across all modes, which corresponds to the conclusion in RQ1, indicating that the features of different files in the QT dataset are relatively stable during training. Additionally, in most cases (Gerrit, JDT, QT projects), models in the origin mode may truncate some important content, leading to suboptimal model performance. In the Platform project, the model in random mode performs the worst, suggesting that if the model's truncation characteristics are not considered and important code information is inadvertently truncated due to the limited length, it may lead to poor model performance. Overall, the proposed Semantic-Align module achieves performance improvements across almost all metrics, especially in classification metrics such as F1 and MCC.

Answer to RQ3: Across multiple datasets, the semantic alignment mode significantly enhances model performance. Particularly, compared to the random input mode, it provides better stability by ensuring consistent input and truncation across multiple training runs. Furthermore, compared to the default mode, semantic alignment efficiently places important content at the beginning of the input sequence, improving the model's learning capability.

VI. RELATED WORK

A. Just-In-Time Defect Prediction Datasets

Before conducting JIT-SDP, constructing a high-quality dataset is a crucial step. Zeng et al. [14] proposed the LApredict dataset. However, the dataset contain outdated commits (2011–2020), which may not reflect current software engineering practices. Additionally, it imposes strict constraints on the scale of code changes, such as a maximum of 10 modified files per commit and a limit of 10 lines of added or removed code per file, potentially limiting the model's learning capability. Ni et al. [8] introduced the Defects4J dataset based on LLTC4J [12], which has been widely used in software defect prediction and localization research. However, Defects4J consists exclusively of Java projects, restricting its applicability in multi-language environments. Furthermore, it stores code change lines using a set structure, meaning that the order of code changes is randomized each time the dataset is loaded. This prevents the dataset from faithfully preserving the original sequence of code modifications in projects, which may impact sequence-based modeling approaches. Keshavarz and Nagappan [29] constructed the ApacheJIT dataset, which, like the Defectors dataset introduced by Mahbub et al. [30], is limited to a single programming language—Java for ApacheJIT and Python for Defectors. ApacheJIT only contains expert features, which limits its applicability in deep learning-based JIT-SDP tasks, particularly in cross-language JIT-SDP research.

In summary, existing JIT-SDP datasets exhibit significant limitations in language coverage, code change storage mechanisms, and the completeness of recorded modifications. Future dataset construction efforts should focus on ensuring data timeliness, language diversity, and preserving the integrity and structure of code changes to support broader JIT-SDP research and applications.

B. Just-In-Time Defect Prediction Framework

In the early stages of JIT-SDP, researchers typically used machine learning techniques to build prediction models. These methods generally relied on expert features, which were defined by experts based on their professional knowledge and experience in understanding defect-introducing commits. Kamei et al. [16] were the first to define 14 expert features in previous research, including five dimensions: the diffusion, size, purpose, history, and programmer experience of code changes. Since then, Kamei's 14 features have been widely adopted in JIT-SDP and have been proven effective [14], [31], [32].

In addition, some researchers incorporated both the code and the commit's textual description into deep learning models. These studies usually focused on the commit messages and the additions or deletions of lines in a commit. Wang et al. [33] used a Deep Belief Network (DBN) architecture to represent source code semantics, and Li et al. [34], [35] used a Convolutional Neural Network (CNN) architecture to learn both semantic and structural features of source code. Deep JIT [17] used a text-oriented Convolutional Neural Network (TextCNN) to automatically extract features from code changes and commit logs as input for classification. CC2Vec [18] learned distributed representations of code changes from log information, capturing the semantic intent of the code changes. However, Zeng et al. [14] argued that in their expanded JIT defect prediction dataset, CC2Vec does not consistently outperform Deep JIT [17]. After experimental validation, they proposed a simple model, LApredict, which only uses the number of added lines in a commit as a feature in a logistic regression classifier.

Building on existing methods, some researchers proposed fusion models combining expert features and semantic features for JIT-SDP. These mainly include two fusion approaches: feature fusion and model fusion. Feature fusion occurs at the feature level, where various features from different modalities are concatenated into one large vector for classification [36], [37]. For instance, Ni et al. [8] proposed JITFine, which fuses expert features and semantic features before tokenization. Similarly, JIT-Smart [13] used the same fusion approach. Model fusion [38], [39] involves independently building multiple models and combining their outputs during the decision phase (e.g., prediction scores). Model fusion can be done using various rules to determine how to combine the outputs of independently trained models [40], such as multiplying, summing, averaging, or weighted averaging all model outputs. For example, Zhou et al. [15] proposed SimCom, which treats JIT defect prediction as a multimodal task and combines prediction scores from a random forest-based machine learning model and a TextCNN-based deep learning model to achieve better performance.

In summary, researchers recommend combining expert features (corresponding to machine learning) and semantic fea-

tures (corresponding to deep learning) for JIT-SDP. Some researchers have already started using pre-trained models, but there are still limitations in their application to JIT-SDP, such as the conflict between the limited input length of pre-trained models and the full length of code changes.

C. Pre-trained Models

In recent years, PTMs like BERT [41] and GPT [42]–[44], which are designed for natural languages (NL), have inspired new ways of representing programming languages (PL). CodeBERT [9] is a dual-modality PTM for code and commit messages, capturing token semantics and context, and has been adopted in JIT-SDP [8], [13]. Wang et al. [10] proposed CodeT5, a unified pre-trained encoder-decoder transformer model that can better leverage code semantics conveyed through identifiers assigned by developers. Lin et al. [11] proposed CCT5, a PTM specifically designed for code changes to better support software maintenance, which also includes JIT-SDP as one of its application tasks.

However, these PTMs are typically constrained by input sequence length limits, which results in truncation when processing long texts (with input lengths restricted to 512 or 1024 tokens). This truncation not only risks losing important information but also affects the model's performance on tasks involving long texts. To address this issue, several solutions have been proposed.

Longformer [45] combined local sliding window attention with global attention, capturing long-range dependencies while maintaining computational efficiency. Similarly, BigBird [46] extended sequence lengths to thousands of tokens by combining random attention, local attention, and global attention, with theoretical guarantees of its universal approximation ability. In addition to improving attention mechanisms, researchers have also explored using structured inputs and external memory to mitigate length limitations. ETC [47] introduced a global-local attention mechanism, dividing inputs into global memory and local sequences to enable more efficient modeling for long-text tasks. Furthermore, Reformer [48] reduced memory consumption through reversible layers and local sensitive hashing (LSH), supporting longer sequence processing.

These studies primarily focus on leveraging techniques to extend the effective length of PTMs to alleviate truncation issues. However, they require specific hardware support. In pre-trained model-based JIT-SDP methods, how can the limited input space be optimized by leveraging the unique characteristics of the JIT-SDP task without requiring additional hardware support, thereby enhancing model performance? Currently, research in this area remains insufficient.

VII. THREATS TO VALIDITY

The conclusions drawn from the method proposed in this paper may not be applicable to other JIT-SDP datasets. However, the experiments in this paper have made efforts to select multiple representative open-source repositories to construct the datasets. In this work, we employ the classic SZZ algorithm, which is currently the most widely used method for obtaining defect labels. However, the SZZ algorithm may have certain errors when labeling. To mitigate this threat, we conduct manual reviews of suspicious labels during the dataset construction process and execute the algorithm only after reaching a consensus on critical steps with other authors.

VIII. CONCLUSION AND FUTURE WORK

Considering that previous datasets are outdated, incomplete, and lack contextual information and file labels, we construct a large-scale dataset, MC4Defect, for JIT-SDP in this paper. We propose JIT-Align, a novel approach designed to address the instability threats caused by forced input truncation in PTMs, motivated by a comprehensive analysis of the impact of different truncation contents on model performance. To evaluate its effectiveness, we compare JIT-Align with four strong baselines across five datasets. Moreover, we conduct ablation studies to analyze the impact of the semantic alignment ranking framework on model performance. The evaluation results demonstrate that JIT-Align surpasses baseline methods in classification performance on nearly all datasets while ensuring model stability by prioritizing code changes that align with the commit message and effectively utilizing the PTM's input capacity.

In future work, we will study and explore the fundamental reasons behind the model's performance improvements through the semantic alignment ranking framework.

ACKNOWLEDGMENT

This work was partially supported by the NSF of China under grants No. 62372174 and No. 62276097, Shanghai Municipal Special Fund for Promoting High Quality Development (No. 2021-GYHLW-01007) , and the Shanghai 2024 Science and Technology Innovation Action Plan Star Cultivation (Sailing Program, No. 24YF2720000).

REFERENCES

[1] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2020.

[2] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 44, p. 5–24, Jan. 2018.

[3] K. H. Dam, T. Pham, S. W. Ng, T. Tran, J. C. Grundy, A. K. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," *ArXiv*, vol. abs/1802.00921, 2018.

[4] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, (New York, NY, USA), p. 18–27, Association for Computing Machinery, 2006.

[5] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," *2012 34th International Conference on Software Engineering (ICSE)*, pp. 200–210, 2012.

[6] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pp. 9–9, 2007.

[7] D. Falessi, S. M. Laureani, J. Çarka, M. Esposito, and D. A. d. Costa, "Enhancing the defectiveness prediction of methods and classes via jit," *Empirical Softw. Engg.*, vol. 28, Jan. 2023.

[8] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: Integrating semantic features with expert features for defect prediction and localization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 672–683, 2022.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Nov. 2020.

[10] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Nov. 2021.

[11] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A code-change-oriented pre-trained model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 1509–1521, 2023.

[12] S. Herbold, A. Trautsch, and Ledel, "A fine-grained data set and analysis of tangling in bug fixing commits," *Empirical Softw. Engg.*, vol. 27, Nov. 2022.

[13] X. Chen, F. Xu, Y. Huang, N. Zhang, and Z. Zheng, "Jit-smart: A multi-task learning framework for just-in-time defect prediction and localization," *Proc. ACM Softw. Eng.*, vol. 1, July 2024.

[14] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: How far are we?," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, p. 427–438, 2021.

[15] X. Zhou, D. Han, and D. Lo, "Simple or complex? Together for a more accurate just-in-time defect predictor," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, p. 229–240, 2022.

[16] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, 2018.

[17] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: An end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the 16th International Conference on Mining Software Repositories*, p. 34–45, 2019.

[18] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, p. 518–529, 2020.

[19] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2018.

[20] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, p. 81–90, 2006.

[21] Jacek, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, p. 1–5, 2005.

[22] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 666–676, 2019.

[23] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, June 2021.

[24] S. Wang, L. L. Minku, and X. Yao, "A systematic study of online class imbalance learning with concept drift," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 10, pp. 4802–4821, 2018.

[25] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[26] C. Van Rijsbergen, *Information retrieval*. Butterworth-Heinemann, 1979.

[27] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[28] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, 2020.

[29] H. Keshavarz and M. Nagappan, "Apachejit: A large dataset for just-in-time defect prediction," in *Proceedings of the 19th international conference on mining software repositories*, pp. 191–195, 2022.

[30] P. Mahbub, O. Shuvo, and M. M. Rahman, "Defectors: A large, diverse python dataset for defect prediction," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 393–397, IEEE, 2023.

[31] C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 369–379, 2021.

[32] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 554–565, 2020.

[33] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 297–308, 2016.

[34] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 318–328, 2017.

[35] Y. Li, "Improving bug detection and fixing via code representation learning," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 137–139, 2020.

[36] H. Gunes and M. Piccardi, "Affect recognition from face and body: early fusion vs. late fusion," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 4, pp. 3437–3443 Vol. 4, 2005.

[37] C. G. M. Snoek, M. Worring, and A. W. M. Smeulders, "Early versus late fusion in semantic video analysis," in *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, p. 399–402, Association for Computing Machinery, 2005.

[38] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, 2018.

[39] S. Yu, T. Falck, A. Daemen, L.-C. Tranchevent, J. A. K. Suykens, B. D. Moor, and Y. Moreau, "L2-norm multiple kernel learning and its application to biomedical data fusion," *BMC Bioinformatics*, vol. 11, pp. 309 – 309, 2010.

[40] C. K. Tantithamthavorn and J. Jiarpakdee, "Explainable ai for software engineering," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1–2, 2021.

[41] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, June 2019.

[42] A. Radford, "Improving language understanding by generative pre-training," 2018.

[43] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[44] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[45] I. Beltagy, A. Cohan, and D. Chen, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[46] M. Zaheer, G. Guruganesh, A. Mehta, J. Angeles, A. Talwalkar, X. Xu, M. Usama, S. Kumar, A. Kamath, D. M. Blei, and C. Re, "Bigbird: Transformers for longer sequences," *arXiv preprint arXiv:2007.14062*, 2020.

[47] J. Ainslie, S. Dewan, J. P. M. H. Wang, T. D. Sanders, E. Shinn, S. M. Smith, and J. S. Zhang, "Etc: Encoding long and structured inputs in transformers," *arXiv preprint arXiv:2006.01324*, 2020.

[48] N. Kitaev, Łukasz Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.