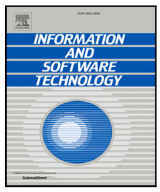
Information and Software Technology 182 (2025) 107706

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



# JIT-CF: Integrating contrastive learning with feature fusion for enhanced just-in-time defect prediction

Xiaolin Ju [a,*], Yi Cao [a], Xiang Chen [a,*], Lina Gong [b], Vaskar Chakma [a], Xin Zhou [a]

[a] School of Artificial Intelligence and Computer Science, Nantong University, Nantong, 226019, Jiangsu, China
[b] School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 211106, Jiangsu, China

ARTICLE INFO

*Keywords:*
Just-in-time defect prediction
Contrastive learning
Feature fusion
CodeBERT

ABSTRACT

**Context:** Just-in-time defect prediction (JIT-DP) is a crucial process in software development that focuses on identifying potential defects during code changes, facilitating early mitigation and quality assurance. Pre-trained language models like CodeBERT have shown promise in various applications but often struggle to distinguish between defective and non-defective code, especially when dealing with noisy labels.
**Objective:** The primary aim of this study is to enhance the robustness of pre-trained language models in identifying software defects by developing an innovative framework that leverages contrastive learning and feature fusion.
**Method:** We introduce JIT-CF, a framework that improves model robustness by employing contrastive learning to maximize similarity within positive pairs and minimize it between negative pairs, thereby enhancing the model's ability to detect subtle differences in code changes. Additionally, feature fusion is used to combine semantic and expert features, enabling the model to capture richer contextual information. This integrated approach aims to improve the identification and resolution of code defects.
**Results:** JIT-CF was evaluated using the JIT-Defects4J dataset, which includes 23,379 code commits from 21 projects. The results indicate substantial performance improvements over seven state-of-the-art baselines, with enhancements of up to 13.9% in F1-score, 8% in AUC, and 11% in Recall@20%E. The study also explores the impact of specific customization enhancements, demonstrating the potential for improved just-in-time defect localization.
**Conclusion:** The proposed JIT-CF framework significantly advances the field of just-in-time defect prediction by effectively addressing the challenges encountered by pre-trained models in distinguishing code defects. The integration of contrastive learning and feature fusion not only enhances the model's robustness but also leads to notable improvements in prediction accuracy, offering valuable insights for future applications in software development.

## 1. Introduction

Defects are an inevitable aspect of software development, often resulting from modifications made during the software's creation and evolution [1,2]. These defects can significantly impact the software's reliability and functionality. Consequently, Software Defect Prediction (SDP) has emerged as an active research topic, focusing on identifying potential defects early in the software development life cycle to enable developers to proactively localize and fix them. To achieve this goal, researchers have developed a variety of defect prediction approaches at different levels of granularity. Coarse-grained approaches typically indicates the defect scope at the file or module level [3–7], while fine-grained approaches seek to pinpoint defects at a more detailed level, such as line number of code [8–19].

Modern software development is inherently iterative, characterized by frequent code changes and updates. This dynamic evolution requires the timely detection of potential defects as they arise. Just-In-Time defect prediction (JIT-DP) focuses on predicting defects at the time code changes are committed, thereby facilitating immediate corrective actions [20–22]. Recently, JIT-DP has emerged as a significant advancement within the Software Defect Prediction domain. State-of-the-art JIT-DP techniques leverage machine learning models to predict

* Corresponding authors.
*E-mail addresses:* ju.xl@ntu.edu.cn (X. Ju), ntucaoyi@outlook.com (Y. Cao), xchencs@ntu.edu.cn (X. Chen), gonglina@nuaa.edu.cn (L. Gong), vaskarchakma7@gmail.com (V. Chakma), xinzhountu@hotmail.com (X. Zhou).

https://doi.org/10.1016/j.infsof.2025.107706
Received 19 November 2024; Received in revised form 2 February 2025; Accepted 24 February 2025
Available online 7 March 2025

whether specific commits will introduce defects, often through the analysis of semantic features or expert features.

Semantic features are derived from the underlying structure and relationships within the code, offering insights into potential defect-prone patterns [23]. These features capture the logical and syntactic elements of code, reflecting how changes may influence software behavior. Conversely, expert features are based on domain expertise and capture characteristics like code change history, developer experience, and modification size. The pioneering work by Kamei et al. [20] defined 14 features across five dimensions: diffusion, size, purpose, code change history, and developer experience. These 14 features have become expert features in the field of JIT-DP research.

Various state-of-the-art research has been proposed to facilitate JIT-DP tasks, making significant progress and achieving promising results [24–27]. For example, deep learning models such as DeepJIT [23] and CC2Vec [25] have shown significant improvements in capturing semantic information from code changes. Additionally, recent work by Ni et al. [28] proposed integrating semantic features with expert features of code changes using a unified learning model, demonstrating enhanced prediction performance.

Despite significant progress in recent years, existing JIT-DP techniques still face several limitations: First, distinguishing between similar code changes remains a significant challenge. Identifying defect-inducing commits is particularly difficult when the code changes that introduce defects closely resemble those that do not. Traditional machine learning models, such as decision trees and support vector machines, often struggle to capture these subtle differences effectively. Even neural network-based models find it challenging to discern fine-grained distinctions between defect-inducing and non-defect-inducing code changes, resulting in diminished prediction performance. Second, existing techniques often fail to capture sufficient code information [29,30]. Defect-prone commits are often influenced by the broader context in which code changes occur, including historical commit patterns and interactions between different code segments. However, most existing techniques focus primarily on isolated code modifications, neglecting the surrounding context. This incomplete comprehension of the code's environment reduces prediction accuracy, particularly within complex software systems. Third, existing JIT-DP techniques are often sensitive to noisy labels. Noisy labels are a prevalent issue in software defect datasets, where mislabeled or incomplete annotations can mislead the model during training. Most existing JIT-DP techniques rely on supervised learning approaches that assume all labels are reliable, making them vulnerable to errors introduced by noisy data. This sensitivity can cause the model to mislabel defect-prone code, thereby reducing the model's overall effectiveness.

To address the limitations of existing JIT-DP techniques, we introduce contrastive learning into the semantic feature extraction stage. Contrastive learning operates by maximizing the similarity between semantically related code changes and minimizing the similarity between unrelated changes [31–34]. This approach allows the model to learn more nuanced representations, making it more robust to subtle code changes and reducing the impact of noisy labels. As a result, the model's predictive performance improves, leading to better identification of defect-inducing commits.

Additionally, we refine the model architecture during the feature fusion stage to enhance context capture. By integrating semantic features derived from the code with expert-crafted features based on domain knowledge, we optimize the fusion process through various configurations of fully connected layers and activation functions. This architectural tuning enables the model to learn higher-level abstractions, capturing both localized code changes and their broader context. Consequently, the model can accurately identify defect-prone areas while understanding the complex relationships between code components, significantly improving the performance and reliability of JIT defect prediction.

In this paper, we propose a novel approach called **JIT-CF**, which enhances the ability of pre-trained model [35,36] to extract semantic features through contrastive learning. To the best of our knowledge, JIT-CF is the first to introduce contrastive learning into the JIT-DP task. Specifically, during the semantic feature extraction phase, we employ contrastive learning to enhance CodeBERT's capability to differentiate between similar code changes. Furthermore, to improve the model's ability to capture context, we implement feature fusion, during which we fine-tune the model and identify an optimal configuration. These strategies significantly improve the performance and reliability of JIT-DP, providing a robust and scalable solution for predicting defect-inducing code modifications.

We also evaluate JIT-CF using the JIT-Defects4J dataset, which comprises 21 open-source projects with 27,391 code changes, providing a comprehensive benchmark for defect prediction. To assess the performance of JIT-CF, we compare it against seven state-of-the-art baselines [23–26,28,37,38] across a range of experimental settings. Our evaluation employs five performance metrics, categorized into effort-agnostic and effort-aware measures, to ensure a comprehensive analysis of prediction performance. Notably, JIT-CF achieves a 13.9% improvement in F1-score over the best baseline, JIT-Fine [28], highlighting its superior capability in identifying defect-inducing code changes. These improvements in prediction performance demonstrate the effectiveness of JIT-CF, confirming its advancement over current state-of-the-art methods.

The key contributions of this paper are as follows:

- We introduce JIT-CF, an innovative framework that incorporates contrastive learning and feature fusion into the JIT-DP task. This approach enhances the model's ability to discern subtle differences in code changes, thereby improving its capacity to differentiate between similar code modifications and effectively mitigate the impact of noisy labels.
- We fine-tune the model during the feature fusion phase, identify an optimal architectural configuration that maximizes performance. This optimal design choice enhances the integration of semantic and expert features, resulting in the best predictive performance in JIT-DP task.
- We conduct an extensive evaluation of the impact of contrastive learning and network architecture optimization on model performance, utilizing metrics such as F1-score and AUC. Our experimental results demonstrate that integrating contrastive learning with feature fusion significantly outperforms seven state-of-the-art approaches.

We provide access to JIT-CF[1] to facilitate and encourage future research in Just-In-Time Defect Prediction.

The rest of this paper is organized as follows. Section 2 outlines the background and motivation for our research. Section 3 details the design of JIT-CF. Section 4 examines the experimental settings, including the comparative baselines and performance metrics considered. Section 5 presents the experimental results. Subsequently, Section 6 discusses various issues and potential threats to validity. Section 7 reviews related work and highlights the novelty of our study. Finally, we conclude our work and suggest directions for future research in Section 8.

## 2. Background and motivation

### 2.1. Just-In-Time defect prediction

Just-In-Time defect prediction (JIT-DP) has attracted growing interest in recent years due to its potential to identify defects at the

[1] https://github.com/ntu-juking/JIT-CF

moment of code commitment, thereby facilitating early detection and resolution. The key goal of JIT-DP is to predict which code changes are likely to introduce defects, enabling developers to take immediate action before the code is merged into the software system. This proactive approach contrasts with traditional defect prediction techniques, which often focus on detecting defects after they have been introduced.

The foundational work in this field by Mockus and Weiss [39] introduced a classifier that utilizes commit history information—such as the number of modified subsystems, changed files, and lines of code added—to identify high-risk commits. Building on this foundation, Kamei et al. [20] proposed 14 change-level features, which have since become recognized as expert features in JIT-DP research for capturing the characteristics of code changes. These features are especially valuable in effort-aware scenarios, where prioritizing changes based on the effort required to resolve them is crucial.

Subsequent studies have aimed to enhance JIT-DP models by refining feature representation and prediction techniques. For instance, Yang et al. [24,40] introduced deep learning-based models to capture more complex relationships in code changes. They developed a method using Deep Belief Networks (DBNs) to extract high-level features [24], and later combined decision trees with ensemble learning to create a more robust predictor [40].

As research in this domain advanced, the focus shifted towards enhancing model performance and addressing practical challenges. For example, Young et al. [19] proposed a deep ensemble approach that optimizes weights across classifiers to boost performance. Meanwhile, Liu et al. [41] introduced an unsupervised model for JIT-DP called Code Changes. Chen et al. [42] framed JIT-DP as a multi-objective optimization problem to identify features that improve predictive performance. McIntosh et al. [15] conducted a longitudinal study involving over 37,000 changes, revealing that longer time intervals between training and testing negatively impact model performance, recommending that at least six months of historical data for training.

Other research has addressed challenges in the JIT-DP application. Wan et al. [43] reviewed current defect prediction tools and surveyed practitioners to identify limitations and future research needs. Cabral et al. [8] tackled issues of validation latency and class imbalance in online JIT-DP using a new sampling technique. More recently, deep learning techniques have been applied to JIT-DP, such as the work by Hoang et al. [23,25], who developed methods that learn representations from both commit messages and code changes, effectively capturing the semantics and context of code modifications.

A typical JIT-DP workflow generally involves feature extraction from code changes, utilizing both semantic features (e.g., code structure and syntactic dependencies) and expert-defined features (e.g., change history, developer experience) are used to train a model. This trained model then predicts the likelihood of a new code change being defect-inducing. Through this process, JIT-DP aims to provide actionable insights to developers, thereby enhancing software quality and reducing the cost associated with fixing defects post-deployment.

### 2.2. Contrastive learning

Contrastive learning is a representation learning technique designed to bring similar samples closer together in the feature space while pushing dissimilar samples farther apart. By minimizing the distance between similar pairs and maximizing the distance between dissimilar pairs, contrastive learning effectively enhances the model's ability to learn discriminative features. This approach has demonstrated considerable success across various domains in recent years, particularly in computer vision and natural language processing. In the field of computer vision, contrastive learning methods have been instrumental in improving image classification, object detection, and representation learning tasks [34,42,44–46]. In natural language processing (NLP), contrastive learning has effectively improved sentence embeddings, text classification, and language understanding by leveraging the similarities and differences between sentence pairs [47–49].

Recent research has increasingly focused on the application of contrastive learning across various software engineering tasks, owing to its effectiveness in learning robust representations. For example, Bui et al. [50] introduced Corder, a contrastive learning framework tailored for software tasks such as code-to-code retrieval, text-to-code retrieval, and code-to-text summarization. By learning the semantic relationships between different code and text samples, Corder achieved notable improvements across these tasks. Similarly, VarCLR [51] employs contrastive learning to capture the semantic representations of variable names, enhancing performance in downstream tasks like variable similarity scoring and correcting variable misspellings. ContraCode [52] further demonstrates the versatility of contrastive learning by generating syntactic variants of JavaScript code through source-to-source compilation. These generated code samples are then used to train a contrastive model, successfully improving tasks such as clone detection, type inference, and code summarization.

Contrastive learning's ability to distinguish fine-grained differences between similar entities while effectively leveraging their context has made it an increasingly popular approach in software engineering research. By learning to better represent code semantics and identify relationships between code changes, contrastive learning holds significant potential for enhancing various tasks, including JIT-DP.

### 2.3. Motivation

JIT-DP plays a critical role in ensuring software quality and reliability. Recent advances in deep learning, particularly in natural language processing (NLP) tasks, have demonstrated the capability of pre-trained models to capture the semantic features of source code [28]. However, despite these advancements, current state-of-the-art JIT-DP techniques still encounter considerable challenges which include difficulty distinguishing between similar code changes, insufficient capture of contextual information, and sensitivity to noisy labels.

The first challenge is to identify differences between similar code changes. This capability is essential for the precise identification of defect-inducing commits. Seemingly minor modifications, such as variable renaming or formatting changes, may not alter the code's logic, yet they can cause existing models to misclassify these changes. Fig. 1 illustrate this issue: the original code snippet (Fig. 1(a)), correctly identified as non-defective, becomes incorrectly classified as defective after a superficial change, like variable renaming (Fig. 1(b)). This sensitivity to minor modifications underscores the limitations of current feature extraction techniques, including pre-trained models such as CodeBERT. These models often concentrate on shallow-level features and encounter difficulties in capturing deeper semantic similarities when confronted with minor syntactic changes.

The second challenge is the insufficient capture of contextual information. Current JIT-DP models often overlook broader context, such as historical commit patterns and interactions between code segments, which can play a critical role in defect introduction. We address this by refining the network architecture during the feature fusion phase, and optimizing the use of fully connected layers and activation functions. This architectural enhancement enables the model to better capture contextual relationships, thereby improving precision in predicting defect-inducing changes.

The third challenge is the sensitivity to noisy labels. Incorrect or incomplete annotations in training datasets often leads to decreased model performance. By refining feature representations and optimizing the learning process, the deep learning approach can become more resilient to noise, thereby enhancing its generalizability across diverse datasets.

To address these challenges, we propose JIT-CF, a novel framework designed to enhance both the representation of code changes and the model's prediction performance in JIT defect prediction.

```python
def add_item_to_cart(cart, item, quantity):
    if item in cart:
        cart[item] += quantity
    else:
        cart[item] = quantity
    return cart

shopping_cart = {}
shopping_cart = add_item_to_cart(shopping_cart, 'apple', 3)
print(shopping_cart)
```

(a) Original code snippet correctly identified as non-defective.

```python
def update_inventory(inventory, product, count):
    if product in inventory:
        inventory[product] += count
    else:
        inventory[product] = count
    return inventory

current_inventory = {}
current_inventory = update_inventory(current_inventory, 'apple', 3)
print(current_inventory)
```

(b) Code snippet after variable renaming incorrectly identified as defective.

**Fig. 1.** An example of defect prediction error caused by overly similar code.

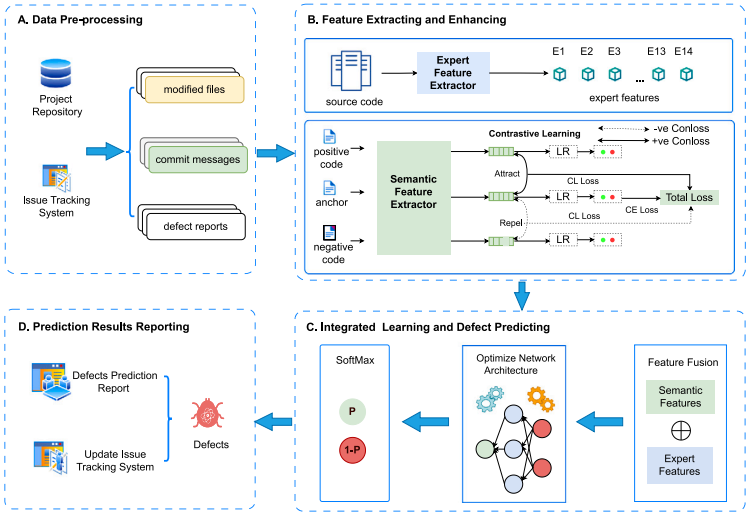



**Fig. 2.** The framework of our approach JIT-CF.

## 3. Approach

Our approach, **JIT-CF** as shown in Fig. 2, incorporates contrastive learning during the semantic feature extraction phase for just-in-time defect prediction. Additionally, JIT-CF optimizes the network architecture with the features fusion to enhance the model's capability in feature representation and improve its robustness. JIT-CF consists of four main steps: ① **Data Pre-processing**: Collect data from code repository and issue tracking system, including code commit history, source code, bug report, etc. ② **Feature Extracting and Enhancing**: Expert features are extracted by adopting the 14 code change-level features defined by Kamei et al. [20]. Concurrently, semantic features are extracted with the pre-trained model CodeBERT. Furthermore, Contrastive learning is applied during the semantic feature extraction phase to enhance feature representation; ③ **Integrated Learning and Defect Predicting**: The optimal fully connected layer is identified by experimenting with various combinations of fully connected layers and activation functions for feature fusion, leading to the final model training; ④ **Prediction Result Reporting**: Create prediction reports of JIT-DP and document the identified defects within issue tracking system. Details of JIT-CF are presented in the following subsections.

### 3.1. Data pre-processing

In the first stage of our approach, we focus on systematically collecting and organizing data from both the code repository and the issue tracking system. This involves gathering comprehensive information, including the code commit history, which encompasses commit messages, added lines, and deleted lines. Additionally, we extract the source code and relevant bug reports. This stage is crucial as it lays the foundation for subsequent analysis by ensuring that all pertinent data is accurately captured and structured. The collected data serves as the basis for further processing and analysis, facilitating a deeper understanding of the code changes and their impact on software quality.

### 3.2. Feature extracting and enhancing

This stage include two main sub-tasks: ① expert features extracting and ② semantic features extracting and enhancing. Together, these subtasks create a comprehensive feature set that combines both structured, expert-driven insights and rich, context-aware semantic representations. This dual approach enhances the overall predictive accuracy and robustness of the model by leveraging the strengths of both expert knowledge and advanced machine learning techniques.

#### 3.2.1. Expert features extracting

Expert feature extraction, involves identifying and extracting predefined code change-level features that have been established by domain experts, providing a structured and quantifiable basis for analysis. These features offer insights into specific aspects of code changes that are considered significant based on prior research and expert knowledge.

In our approach, we utilized the 14 expert features proposed by Kamei et al. [20], which capture various aspects of code changes, including diffusion, size, purpose, history, and developer experience. As detailed in Table 1, these features are selected based on their well-established relevance and effectiveness in the field of JIT defect prediction. For instance, diffusion features (e.g., NS, ND, NF) reflect the scope and complexity of changes, while size features (e.g., LA, LD) quantify the impact of changes on the codebase. Additionally, history features (e.g., AGE, NUC) capture the frequency and stability of changes, and developer experience features (e.g., EXP, SEXP) reflect the familiarity and potential risk associated with developers' modifications.

#### 3.2.2. Semantic features extracting and enhancing

The extraction and enhancement of semantic features are crucial for JIT-DP due to their ability to capture the nuanced and contextual

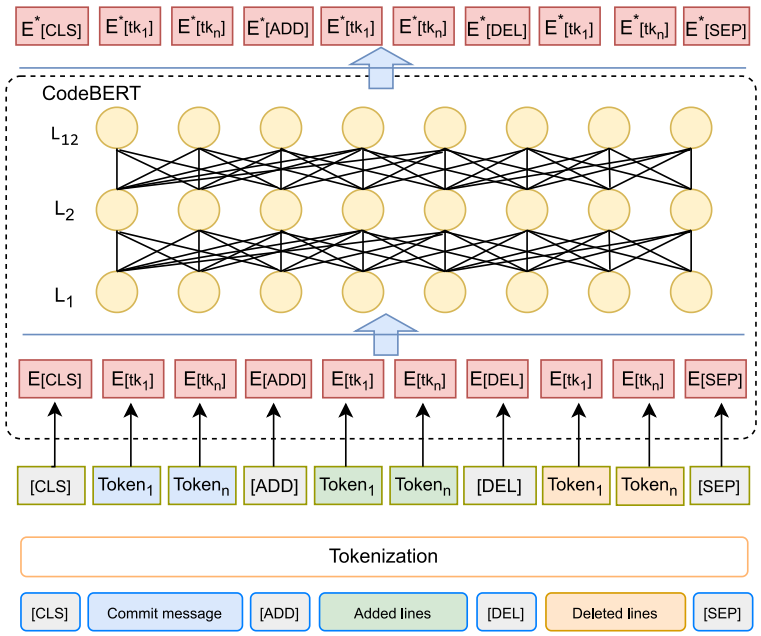



**Fig. 3.** Feature Extraction via CodeBERT. The figure illustrates the process by which code tokens and related metadata are processed through multiple layers within the CodeBERT to generate feature embeddings utilized for JIT-DP.

**Table 1**
The 14 expert features used in our study.

| Name | Description | Dimension |
|---|---|---|
| NS | Number of modified subsystems | Diffusion |
| ND | Number of modified directories | Diffusion |
| NF | Number of modified files | Diffusion |
| Entropy | Distribution of modified code across files | Diffusion |
| LA | Lines of code added | Size |
| LD | Lines of code deleted | Size |
| LT | Lines of code in file before change | Size |
| FIX | Indicates if the change is a defect fix | Purpose |
| NDEV | Number of developers who modified the files | History |
| AGE | Average time between last and current change | History |
| NUC | Number of unique changes to the files | History |
| EXP | Developer's overall experience | Experience |
| REXP | Developer's recent experience | Experience |
| SEXP | Developer's experience in the specific subsystem | Experience |

information embedded within code changes. Unlike expert features, which rely on predefined metrics, semantic features delve into the deeper meanings and relationships within the code, enabling a more comprehensive understanding of the code changes. Our approach leverages CodeBERT for semantic feature extraction to enhance the analysis of software code. CodeBERT, a transformer-based model pre-trained on a large corpus of code and natural language, excels at understanding the intricate relationships between code tokens [28]. By utilizing CodeBERT, we can extract semantic features that capture the deeper meaning and context of the code, beyond what traditional syntactic analysis can achieve.

The details of the semantic features extractor within JIT-CF are shown in Fig. 3, which incorporates various types of information, including commit message, added lines, deleted lines, etc. The term "Commit message" refers to the description of the submitted commit, whereas "added lines" and "deleted lines" represent the lines added and deleted in the commit, respectively.

The input to the CodeBERT model consists of a sequence of tokens derived from the source code, commit messages, and code changes. The tokenization process is critical, as it systematically handles various components of the code, such as added or deleted lines, by differentiating them with specific token markers. For instance, newly added lines might be tagged with a unique token, distinguishing them from modified or deleted lines. This nuanced handling ensures that the structural and semantic integrity of the code is preserved during feature extraction.

Once tokenized, these sequences are transformed into embeddings—high-dimensional vector representations that encode not only the syntactical but also the contextual information of each token. CodeBERT processes these embeddings through multiple transformer layers. These layers, designed with self-attention mechanisms, enable the model to capture intricate dependencies and relationships across different parts of the code. Consequently, the generated feature vectors provide a rich, semantically informed representation of the code, which is valuable for downstream tasks.

These high-dimensional feature vectors serve as the foundational input to our contrastive learning approach. By utilizing CodeBERT's extracted features, the model can focus on the most pertinent aspects of the code, such as subtle changes that might indicate potential defects. This focus is particularly crucial for JIT defect prediction, where the goal is to accurately identify defects based on recent changes in the code repository.

In the original semantic feature extraction phase, CodeBERT extracts semantic features by processing the textual content of the source code. Although CodeBERT is effective at capturing both syntactic and semantic aspects of the code, it still has certain limitations when handling subtle code changes or semantically similar code snippets. For instance, subtle changes like variable renaming or slight formatting adjustments do not affect the logic of the code, but CodeBERT, without additional handling, may interpret these changes as significant semantic differences, leading to incorrect classification. This sensitivity to superficial modifications reduces the performance of JIT-DP, hindering the model's ability to accurately predict defects.

To address this issue, we introduce a contrastive learning strategy aimed at enhancing the robustness and discriminative power of feature representations. Contrastive learning works by maximizing the similarity between positive samples and minimizing the similarity between

negative samples. Specifically, in our approach, we compare code snippets with the same defect labels and those with different defect labels, guiding the model to learn more discriminative features while ignoring irrelevant shallow-level changes.

Given the specific characteristics of JIT-DP, we adapt the standard contrastive learning process. Contrastive learning typically operates based on pairs of positive and negative samples, and in the JIT-DP context, code changes are the core objects being compared. For each code sample, we define a positive set comprising code changes with the same defect status and a negative set consisting of code changes with different defect statuses.

**Algorithm 1:** Contrastive learning for JIT-CF in code commits

**Input:** Code commits $C = \{C_1, C_2, ..., C_N\}$, Labels $y = \{y_1, y_2, ..., y_N\}$, Temperature $\tau$
**Output:** Optimized Code Commit Embeddings $\mathbf{H}$

**for** *each code commit* $C_i$ **do**
  Encode the input code commit $C_i$ using CodeBERT to obtain embedding $\mathbf{H}_i$;
  Define positive set $P_i = \{C_p \mid y_p = y_i\}$ based on commits with the same defect status;
  Define negative set $A_i = \{C_n \mid y_n \neq y_i\}$ based on commits with different defect statuses;
  **for** *each positive commit* $C_p \in P_i$ **do**
    Compute similarity between embeddings $\mathbf{H}_i$ and $\mathbf{H}_p$: $\mathbf{z}_i \cdot \mathbf{z}_p$;
  **for** *each negative commit* $C_n \in A_i$ **do**
    Compute similarity between embeddings $\mathbf{H}_i$ and $\mathbf{H}_n$: $\mathbf{z}_i \cdot \mathbf{H}_n$;
  Compute the contrastive loss $L_{\text{contrast}}$ using the similarities between positive and negative commits;
  Update model parameters using backpropagation to minimize the contrastive loss $L_{\text{contrast}}$;
**return** Optimized code commit embeddings $\mathbf{H}$;

Algorithm 1 describes how we apply contrastive learning to enhance CodeBERT's semantic feature extraction process for JIT-DP. In this algorithm, each input code commit is denoted as $C_i$, where $y_i$ represents its associated label indicating whether the commit is non-defective ($y_i = 0$) or defective ($y_i = 1$). The model first processes $C_i$ using CodeBERT to generate an embedding $\mathbf{H}_i$, which captures the syntactic and semantic features of the code commit. For each anchor code commit $C_i$, a positive set $P_i$ is defined, consisting of all other commits that share the same defect label as $C_i$. This means that these commits either all introduce defects or all do not introduce defects, thus forming the set of positive examples. On the other hand, the negative set $A_i$ is composed of commits that have a different defect label from $C_i$, representing cases where the commits either introduce defects while $C_i$ does not, or vice versa.

Once the positive and negative sets $P_i$ and $A_i$ are defined for each anchor commit, the algorithm computes the similarities between the embedding $\mathbf{H}_i$ of the anchor commit and the embeddings of the positive samples $\mathbf{H}_p$ for all $p \in P_i$, using the dot product of their feature vectors $\mathbf{z}_i \cdot \mathbf{z}_p$. Similarly, the similarities between $\mathbf{H}_i$ and the embeddings of the negative samples $\mathbf{H}_n$ for all $n \in A_i$ are computed. The goal is to maximize the similarity between the anchor commit and positive samples while minimizing the similarity with negative samples. This is achieved through the supervised contrastive loss function, which is defined as:

$$L_{\text{sup}} = \frac{1}{N} \sum_i \frac{1}{|P_i|} \sum_{p \in P_i} \log \frac{\exp(\mathbf{z}_i \cdot \mathbf{z}_p / \tau)}{\sum_{n \in A_i} \exp(\mathbf{z}_i \cdot \mathbf{z}_n / \tau)} \tag{1}$$

where $\mathbf{z}_i$ and $\mathbf{z}_p$ represent the normalized embeddings of the anchor code commit $C_i$ and a positive sample $C_p$, respectively. The numerator encourages the model to maximize the similarity between the anchor commit and positive samples, while the denominator minimizes the similarity between the anchor commit and negative samples $C_n$. The temperature parameter $\tau$ controls the sharpness of the softmax distribution, allowing the model to fine-tune its sensitivity to variations in the embeddings.

This contrastive learning approach, which leverages the comparison of code commits based on their defect status, significantly enhances the model's robustness and discriminative performance. By focusing on the true semantic differences related to defects, rather than superficial syntactic changes, JIT-CF can accurately differentiate between non-defective and defective code commits. This results in more reliable defect predictions, even when minor code modifications that do not affect functionality are present.

### 3.3. Integrated learning and defect predicting

After enhancing the semantic features through contrastive learning, the next step in our JIT-CF framework is to integrate expert features. To integrate the semantic features (768-dimensional vectors) with the lower-dimensional expert features (14-dimensional vectors), we expand the expert features to match the dimensionality of the semantic features. We achieve this expansion by using a fully connected layer that transforms the expert feature vector $\mathbf{V}_{\text{EF}}$ into a higher-dimensional representation. The resulting expanded expert feature vector is then concatenated with the semantic feature vector $\mathbf{V}_{\text{SF}}$ to form a combined feature vector $\mathbf{V}_F$.

After combining the features, we enhance the representation through fully connected layers to capture complex interactions between semantic and expert features. To identify the optimal network architecture for this feature fusion, we systematically tuned the model by experimenting with twelve different combinations of fully connected layers (varying between 1 to 3 layers) and activation functions (ReLU, GELU, ELU, and Leaky ReLU). Each combination was thoroughly evaluated to identify the best-performing configuration that balances feature learning and prediction accuracy.

This fine-tuning process, comprising extensive experimentation across multiple network depths and non-linear transformations, is crucial in enhancing the model's capacity to learn intricate patterns within the combined feature space. By carefully selecting the optimal architecture, we significantly improve JIT-CF's performance, ensuring that it can robustly learn from both semantic and expert-defined features to predict defect-prone code changes effectively.

### 3.4. Prediction result reporting

In the final stage of our approach, we focus on synthesizing and communicating the outcomes of the JIT-DP process. This involves generating comprehensive prediction reports that detail the findings and insights derived from the analysis. These reports are meticulously crafted to provide clear and actionable information about potential defects. Furthermore, the identified defects are systematically documented within the issue tracking system, ensuring that they are integrated into the ongoing workflow for resolution. This stage is essential for closing the loop between prediction and action, enabling teams to address potential issues proactively and maintain high software quality.

## 4. Experimental evaluation

### 4.1. Research questions

We aim to evaluate JIT-CF by answering the following four research questions:

**RQ1: How effective is JIT-CF in JIT-DP?**

To answer this question, we compare JIT-CF with seven state-of-the-art approaches for JIT-DP. The objective is to assess the model's ability to identify potential defects in code changes before they are merged

**Table 2**
Details of Code Commits in JIT-Defects4J.

| Java Project | Timeframe | #BC | #CC | % Ratio (Bugs/ALL) |
|---|---|---|---|---|
| ant-ivy | 2005-06-16 - 2018-02-13 | 332 | 1,439 | 18.75% (332/1,771) |
| commons-bcel | 2001-10-29 - 2019-03-12 | 60 | 765 | 7.27% (60/825) |
| commons-beanutils | 2001-03-27 - 2018-11-15 | 37 | 574 | 6.06% (37/611) |
| commons-codec | 2003-04-25 - 2018-11-15 | 36 | 725 | 4.73% (36/761) |
| commons-collections | 2001-04-14 - 2018-11-15 | 50 | 1,773 | 2.74% (50/1,823) |
| commons-compress | 2003-11-23 - 2018-11-15 | 178 | 1,452 | 10.92% (178/1,630) |
| commons-configuration | 2003-12-23 - 2018-11-15 | 155 | 1,683 | 8.43% (155/1,838) |
| commons-dbcp | 2001-04-14 - 2019-03-12 | 58 | 979 | 5.59% (58/1,037) |
| commons-digester | 2001-05-03 - 2018-11-16 | 19 | 1,067 | 1.76% (19/1,086) |
| commons-io | 2002-01-25 - 2018-11-16 | 73 | 1,069 | 6.39% (73/1,142) |
| commons-jcs | 2002-04-07 - 2018-11-15 | 88 | 743 | 10.59% (88/831) |
| commons-lang | 2002-07-19 - 2018-10-10 | 146 | 2,823 | 4.92% (146/2,969) |
| commons-math | 2003-05-12 - 2018-02-15 | 335 | 3,691 | 8.32% (335/4,026) |
| commons-net | 2002-04-03 - 2018-11-14 | 117 | 1,004 | 10.44% (117/1,121) |
| commons-scxml | 2005-08-17 - 2018-11-16 | 47 | 497 | 8.64% (47/544) |
| commons-validator | 2002-01-06 - 2018-11-19 | 36 | 562 | 6.42% (36/598) |
| commons-vfs | 2002-07-16 - 2018-11-19 | 114 | 996 | 10.27% (114/1,110) |
| giraph | 2010-10-29 - 2018-11-21 | 163 | 681 | 19.31% (163/844) |
| gora | 2010-10-08 - 2018-06-18 | 39 | 514 | 7.05% (39/553) |
| opennlp | 2008-09-28 - 2018-06-18 | 91 | 995 | 8.38% (91/1,086) |
| parquet-mr | 2012-08-31 - 2018-07-01 | 158 | 962 | 14.11% (158/1,120) |
| ALL | | **2,332** | **24,987** | **8.54% (2,332/27,319)** |

into code repository. This evaluation will help determine whether JIT-CF provides significant improvements over existing methods.

**RQ2: How does contrastive learning influence the performance of JIT-CF?**

In JIT-CF, contrastive learning is fundamental, aiming to improve the model's capacity to distinguish between similar and dissimilar code changes. This research question investigates the impact of semantic features, enhanced through contrastive learning, on the model's performance.

**RQ3: What impact does the optimization of model architecture, incorporating feature fusion, have on JIT-CF?**

For JIT-CF, optimizing the model architecture incorporating feature fusion is crucial for enhancing its ability to capture complex patterns in data. This RQ examines how modifications in the configuration of layers and activation functions affect performance. By analyzing these optimizations, we aim to identify the optimal design strategy to balance model complexity and prediction performance, thereby providing valuable insights for future JIT-DP models.

### 4.2. Datasets

For our experiments, we utilize the JIT-Defects4J [28] dataset, which serves as a comprehensive and widely-adopted benchmark for JIT defect prediction research. Specifically, the dataset includes 27,319 code commit records sourced from 21 well-maintained Java open-source projects, spanning diverse domains such as libraries, utilities, and web frameworks. Among these commits, 2332 are labeled as defective, while the remaining 24,987 are labeled as non-defective. The details of the dataset related to defect prediction are shown in Table 2.

The JIT-Defects4J [28]dataset is meticulously curated to capture real-world software development patterns, providing not only commit histories but also detailed defect information that reflects actual software bugs encountered during the development process. This dataset includes a rich set of features, such as code metrics (e.g., lines of code changed, file diffusion), project-level attributes, and developer activities. These features make it highly suitable for training deep learning models that demand an in-depth understanding of both code structure and commit metadata.

### 4.3. Evaluation metrics

To evaluate the performance of JIT-CF, we employ two categories of evaluation metrics: **effort-agnostic performance measures** and **effort-aware performance measures**. These metrics offer a comprehensive understanding of the model's performance, considering both general prediction accuracy and the practical effort required in real-world scenarios.

*Effort-agnostic performance measures.* focus on the overall prediction quality without considering the associated effort for identifying defects. The key metrics in this category are F1-score and AUC.

**F1-score** is widely used in classification tasks, particularly for imbalanced datasets. It is the harmonic mean of precision and recall, balancing the trade-off between the two. The F1-score is defined as:

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4}$$

where $TP$, $FP$, and $FN$ represent true positives, false positives, and false negatives, respectively.

**Area Under the ROC Curve (AUC)** measures the area under the Receiver Operating Characteristic (ROC) curve, which plots the true positive rate (TPR) against the false positive rate (FPR) at various thresholds. AUC provides a scalar value that summarizes the model's performance across all thresholds:

$$\text{AUC} = \int_0^1 \text{TPR}(x)\, dx \tag{5}$$

*Effort-aware performance measures.* incorporate the practical cost and effort associated with defect detection, making them more relevant to real-world scenarios where reviewing code changes requires significant effort. The key metrics in this category are Recall@20%Effort, Effort@20%Recall, and Popt.

**Recall@20%Effort (R@20%E)** evaluates the proportion of defects identified within the top 20% of the riskiest code changes, as predicted

by the model. It is calculated as:

$$\text{R@20\%E} = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{n} TP_i} \tag{6}$$

where $k$ is the number of changes constituting 20% of the total effort, and $n$ is the total number of changes.

**Effort@20%Recall (E@20%R)** indicates the effort required to detect 20% of all defects. It reflects the percentage of code that needs to be inspected to achieve this recall level. The formula is:

$$\text{E@20\%R} = \frac{\sum_{i=1}^{m} \text{LOC}_i}{\sum_{i=1}^{N} \text{LOC}_i} \tag{7}$$

where $m$ is the number of changes needed to achieve 20% recall, $\text{LOC}_i$ represents the lines of code in change $i$, and $N$ is the total number of changes.

**Popt** is on the basic of the concept of the Alberg diagram [53] which evaluates the model's performance in prioritizing defective changes by comparing the effort spent reviewing changes the model prioritized to an optimally prioritized effort. It is calculated as:

$$\text{Popt} = 1 - \frac{\sum_{i=1}^{n}(\text{Rank}_{\text{model}}(i) - \text{Rank}_{\text{optimal}}(i))^2}{\sum_{i=1}^{n}(\text{Rank}_{\text{worst}}(i) - \text{Rank}_{\text{optimal}}(i))^2} \tag{8}$$

where, $\text{Rank}_{\text{model}}(i)$ is the rank assigned to change $i$ by the model, $\text{Rank}_{\text{optimal}}(i)$ is the optimal rank, and $\text{Rank}_{\text{worst}}(i)$ is the worst possible rank.

By considering both effort-agnostic and effort-aware metrics, we obtain a more comprehensive evaluation of our defect prediction model's performance, addressing its prediction accuracy and the real-world effort required for defect identification.

### 4.4. Baseline methods

We evaluate our proposed model against seven state-of-the-art methods: LApredict, Deeper, DeepJIT, CC2Vec, Yan et al.'s work, JITLine, and JIT-Fine. The selection of these baselines is driven by their established effectiveness and relevance in the field of just-in-time defect prediction. Each of these methods represents a significant advancement in leveraging machine learning techniques to predict software defects based on code changes and commit histories.

**(1) LApredict** [26]: LApredict uses Logistic Regression on handcrafted features from commit messages and code changes, such as lines of code added or deleted. Its simplicity and effective feature engineering make it a solid baseline.

**(2) Deeper** [24]: Deeper employs Convolutional Neural Networks (CNNs) to learn feature representations from code changes. It processes code diffs as token sequences, capturing local patterns with multiple convolutional layers.

**(3) DeepJIT** [23]: DeepJIT combines Long Short-Term Memory (LSTM) networks and CNNs to capture sequential and spatial features from commit messages and code changes. It processes each separately before combining their representations.

**(4) CC2Vec** [25]: CC2Vec encodes code changes into vector representations using hierarchical attention to focus on relevant parts. It generates context-aware embeddings for defect prediction.

**(5) Yan et al.'s Work** [38]: Yan et al. propose a hybrid model that integrates traditional machine learning with deep learning. It uses handcrafted features from code changes and commit messages as inputs to a deep neural network.

**(6) JITLine** [37]: JITLine integrates both syntactic and semantic features by utilizing AST-based features and word embeddings extracted from commit messages. This comprehensive approach enhances its predictive performance, making it highly effective in identifying software defects.

**(7) JIT-Fine** [28]: JIT-Fine extends JITLine by integrating expert-defined features with semantic features from pre-trained models like CodeBERT. It uses a multi-layer fully connected network and attention mechanisms for improved prediction performance.

**Table 3**
Defect prediction of JIT-CF compared against seven baselines.

| Methods | F1-score | AUC | R@20%E | E@20%R | Popt |
|---|---|---|---|---|---|
| LApredict | 0.059 | 0.694 | 0.625 | 0.020 | 0.814 |
| Yan et al. | 0.062 | 0.675 | 0.615 | 0.022 | 0.819 |
| Deeper | 0.246 | 0.682 | 0.638 | 0.021 | 0.827 |
| DeepJIT | 0.293 | 0.775 | 0.676 | 0.014 | 0.860 |
| CC2Vec | 0.248 | 0.791 | 0.676 | 0.014 | 0.861 |
| JITLine | 0.261 | 0.802 | 0.705 | 0.015 | 0.883 |
| JIT-Fine | 0.431 | 0.881 | 0.773 | 0.010 | 0.927 |
| JIT-CF | **0.491** | **0.896** | **0.831** | **0.010** | **0.944** |

### 4.5. Experimental settings

The Experiment was conducted on a server equipped with an NVIDIA GeForce RTX 3090 GPU. In our experiments, we adopted the same experimental settings as described in JIT-Fine [28]. We partitioned the JIT-Defects4J dataset into disjoint training, validation, and test sets, using a random split ratio of 8:1:1. Our model leverages CodeBERT with a maximum input sequence length of 512 tokens. We set the maximum number of training epochs to 50, starting with an initial learning rate of 5e−4, which gradually increased from 0 during the warm-up phase. For the contrastive learning component, we set the temperature parameter ($\tau$) to 0.7 and employed a batch size of 32 per GPU with gradient accumulation steps of 32. To ensure efficient training and prevent overfitting, we implemented early stopping with a patience of 10 epochs.

## 5. Experiment results

### 5.1. RQ1. Effectiveness of JIT-CF

To demonstrate the effectiveness of JIT-CF, we compared JIT-CF with seven baselines on JIT-Defects4J. The experimental results are presented in Table 3.

From Table 3, we observe that our proposed method, JIT-CF, outperforms all the baselines across all evaluated metrics. JIT-CF achieves the highest performance in terms of F1-score, AUC, R@20%E, and Popt, while maintaining a competitive results for E@20%R. Specifically, JIT-CF improves the F1-score by 13.9% relative to the current best baseline, JIT-Fine. The corresponding relative improvement in AUC is 0.65%, demonstrating the model's enhanced ability to distinguish between defective and non-defective changes. Furthermore, JIT-CF increases the R@20%E by 7.29%, highlighting its efficiency in identifying a larger proportion of actual defects within the top 20% of predictions. Although the E@20%R remains consistent with the baseline, the Popt metric shows a relative improvement of 1.86%, indicating a more optimized effort in inspecting the most defect-prone code changes.

In essence, our results indicate that the JIT-CF surpasses existing works that utilize traditional and deep learning-based methods for JIT-DP. Numerous prior studies have demonstrated the effectiveness of various machine learning and deep learning models for defect prediction. However, the results presented in Table 3 demonstrate that through the integration of contrastive learning and feature fusion, our method outperforms all other baselines on the JIT-Defects4J dataset.

Several factors contribute to this performance enhancement. First, contrastive learning significantly enhances the feature representations by encouraging the model to distinguish between subtle differences in code changes. This enhancement is crucial in JIT defect prediction task, where minor variations can indicate the presence of defects. Second, the multi-layer fully connected architecture allows the model to capture complex interactions between features, further boosting its prediction performance. These combined factors enable JIT-CF to effectively outperform other methods.

**Table 4**
Impact of Semantic and Expert Features with and without Contrastive Learning on Model Performance.

| Setting | F1-score | AUC | R@20%E | E@20%R | Popt |
|---|---|---|---|---|---|
| EF | 0.230 | 0.661 | 0.632 | 0.020 | 0.825 |
| SF | 0.375 | 0.846 | 0.731 | 0.015 | 0.901 |
| SF+CL | 0.396 | 0.851 | 0.739 | 0.014 | 0.911 |
| EF+SF | 0.431 | 0.881 | 0.773 | 0.010 | 0.927 |
| EF+SF+CL | **0.491** | **0.896** | **0.831** | **0.010** | **0.944** |

**Answer to RQ1:** JIT-CF outperforms all baselines in terms of F1-score, AUC, R@20%E, and Popt. The F1-score of JIT-CF is 13.9% higher than the best baseline method, JIT-Fine, indicating its superior performance in JIT-DP.

### 5.2. RQ2. Effectiveness of contrastive learning

To address RQ2, we conducted ablation experiments to evaluate the effectiveness of contrastive learning in JIT-CF. We compared five experimental configurations: Expert Features (EF), Semantic Features (SF), Semantic Features with Contrastive Learning (SF+CL), Expert Features and Semantic Features (EF+SF), and Expert Features and Semantic Features with Contrastive Learning (EF+SF+CL). The results are detailed in Table 4.

The EF+SF+CL configuration consistently outperforms other setups across all metrics. When contrastive learning is added to SF alone, the F1-score improves by 5.6%, AUC increases slightly, and R@20%E rises, indicating a higher capture of true positives. Although E@20%R decreases marginally, Popt improves, suggesting better prioritization of defect-prone changes.

Comparing EF to EF+SF, the inclusion of semantic features leads to a significant performance boost: the F1-score increases by 87.4%, from 0.230 to 0.431, and AUC improves by 33.2%. This highlights the importance of integrating expert and semantic features for capturing more meaningful information about the code changes.

In the comparison EF+SF and EF+SF+CL, the impact is even more pronounced: F1-score rises by 13.9%, from 0.431 to 0.491, demonstrating a significant increase in defect identification capability. AUC improves by 1.7%, indicating better separation between defective and non-defective changes. R@20%E also sees a 7.5% improvement, and Popt rises from 0.927 to 0.944, confirming the efficiency of contrastive learning in optimizing defect prioritization.

To further investigate the impact of contrastive learning, we performed a t-SNE visualization of CodeBERT and CodeBERT with contrastive learning (CodeBERT_CL). t-SNE is a widely used dimensionality reduction technique that maps high-dimensional data to a lower-dimensional space (typically 2D or 3D) while preserving local similarities between data points. This makes it particularly useful for visualizing complex data distributions and identifying clusters or separations in the data. As shown in Fig. 4. in these visualizations, red points represent defective instances, and blue points represent non-defective instances.

In Fig. 4(a), which representing CodeBERT without contrastive learning, there is a significant overlap between defective and non-defective instances, particularly in the boundary areas. This suggests that CodeBERT alone struggles to clearly distinguish between the two classes. Conversely, Fig. 4(b) demonstrates that the introduction of contrastive learning leads to a much clearer separation between defective and non-defective instances, with fewer overlapping points. This improved separation highlights how contrastive learning enhances the model's ability to discriminate between risky and safe code changes.

The effectiveness of contrastive learning in enhancing the model's discriminative power can be attributed to its ability to better represent code features. By effectively separating defective and non-defective instances in the feature space, contrastive learning enables the model to focus on underlying structural and semantic differences, such as subtle changes caused by variable renaming or code refactoring. This fine-grained feature extraction capability allows JIT-CF to perform exceptionally well in handling scenarios where the code changes appear similar on the surface but differ logically, thereby significantly improving defect detection accuracy.

**Table 5**
Performance of using different fully connected layers and activation functions.

| Layers | Activation Function | F1-score | AUC | R@20%E | E@20%R | Popt |
|---|---|---|---|---|---|---|
| 1 layer | ReLU | 0.424 | 0.847 | 0.789 | 0.011 | 0.927 |
| 1 layer | GELU | 0.425 | 0.842 | 0.770 | 0.012 | 0.920 |
| 1 layer | Leaky ReLU | 0.420 | 0.845 | 0.774 | 0.012 | 0.922 |
| 1 layer | ELU | 0.428 | 0.843 | 0.772 | 0.012 | 0.921 |
| 2 layers | ReLU | **0.491** | **0.896** | **0.831** | **0.010** | **0.944** |
| 2 layers | GELU | 0.454 | 0.887 | 0.793 | 0.012 | 0.935 |
| 2 layers | Leaky ReLU | 0.466 | 0.882 | 0.804 | 0.010 | 0.936 |
| 2 layers | ELU | 0.447 | 0.889 | 0.802 | 0.012 | 0.935 |
| 3 layers | ReLU | 0.431 | 0.855 | 0.775 | 0.011 | 0.910 |
| 3 layers | GELU | 0.435 | 0.850 | 0.768 | 0.011 | 0.908 |
| 3 layers | Leaky ReLU | 0.428 | 0.852 | 0.770 | 0.011 | 0.909 |
| 3 layers | ELU | 0.426 | 0.851 | 0.769 | 0.011 | 0.908 |

**Answer to RQ2:** Contrastive learning enhances JIT-CF's ability to differentiate between defective and non-defective code changes by improving its capacity to capture subtle structural and semantic differences. This leads to more effective identification of defective code changes.

### 5.3. RQ3: Effectiveness of model architecture optimization

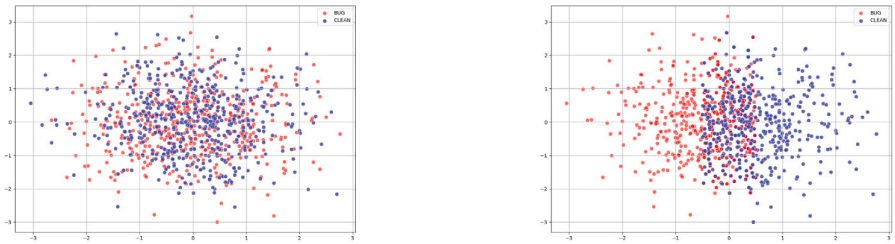
To explore the impact of model architecture optimization on the performance of the JIT-CF model, we investigated two main factors: the number of fully connected layers and the choice of activation functions. Our goal was to identify an architecture that effectively balances model complexity and performance across key metrics, including F1-score, AUC, R@20%E, E@20%R, and Popt.

We experimented with a total of 12 different combinations, consisting of three configurations for the number of layers (1 layer, 2 layers, and 3 layers) paired with four different activation functions (ReLU, GELU, Leaky ReLU, and ELU). The detailed performance results are presented in Table 5.

From Table 5, it is evident that performance varies significantly depending on both the number of layers and the activation function used. A key trend observed is that increasing the depth of the model (from 1 to 3 layers) generally improves performance initially, but can lead to diminishing returns and even degradation in some cases. Notably, the 2-layer configurations consistently outperform both the 1-layer and 3-layer models, suggesting that two layers strike the optimal balance between model capacity and generalization ability. Among these, the configuration with ReLU as the activation function achieves the best overall results, achieving an F1-score of 0.491, an AUC of 0.896, and a Popt of 0.944.

The performance advantage of using 2 layers with ReLU is evident across all metrics. While the 1-layer models suffer from insufficient capacity to learn complex patterns in the code changes, the 3-layer models introduce additional depth that does not translate into improved performance, likely due to overfitting and increased training difficulty. This effect is visible in the drop in metrics such as F1-score and AUC when moving from 2 layers to 3 layers across different activation functions.

The superior performance of the 2-layer model with ReLU can be attributed to its balanced architecture. With two layers, the model has enough capacity to capture complex interactions and patterns within



(a) t-SNE of CodeBERT without contrastive learning

(b) t-SNE of CodeBERT enhanced with contrastive learning

**Fig. 4.** t-SNE visualization comparison between CodeBERT and CodeBERT enhanced with contrastive learning.

the feature space without becoming overly complex. ReLU's effectiveness as the activation function lies in its simplicity and efficiency. It introduces non-linearity without complicating the gradient flow during training, helping the model learn effectively from the data. ReLU also mitigates issues like vanishing gradients, which is crucial for models with deeper architectures.

**Answer to RQ3:** The best configuration for JIT-CF is a two-layer architecture with ReLU activation. This setup maximizes model performance key metrics, achieving the optimal balance between model complexity and generalization capability, thereby enhancing JIT-DP.

## 6. Discussion

### 6.1. How does JIT-CF perform in Just-In-Time defect localization?

Our approach, similar with JIT-Fine, employs a unified model to concurrently perform JIT defect prediction and defect localization. CodeBERT, as a pre-trained model for code representation, plays a crucial role in this process by extracting semantic and contextual features from code changes, enabling it to rank lines of code based on their likelihood of being defective. The embeddings generated by CodeBERT allow the model to identify potential defect-prone areas within a commit effectively.

Furthermore, the application of contrastive learning enhances these feature representations, significantly improving the model's ability to differentiate between defective and non-defective lines, even in cases of minor code variations. This enhances the precision of defect localization, as the model learns to focus on relevant differences, resulting in better identification and ranking of defect-prone lines within code changes.

We evaluate its performance of our approach against three baseline methods: JITLine [37], Yan et al. [38], and JIT-Fine [28]. The evaluation employs several metrics, including Top-5 and Top-10 accuracy, Recall@20%Effort$_{line}$ (R@20%E$_{line}$), Effort@20%Recall$_{line}$ (E@20%R$_{line}$), and Initial False Alarm$_{line}$ (IFA$_{line}$). The results are presented in Table 6.

Top-5 and Top-10 accuracy metrics assess the model's capability to accurately identify defects within its top 5 and top 10 predictions, respectively. R@20%E$_{line}$ evaluates the proportion of defective lines that can be found with the top 20% of lines of code ranked by risk, given a certain effort. A higher value in this metric indicates more accurate identification of defect-prone lines. E@20%R$_{line}$ measures the effort required to identify 20% of the actual defective lines, typically expressed in LOC. A lower value means that the developers can locate defect on these lines with reduced effort. IFA$_{line}$ reflects the amount of code that needs to be inspected before encountering the first false alarm, with a lower value indicating fewer false positives early in the review process.

**Table 6**
Defect localization of JIT-CF compared against baselines.

| Methods | Top-5 | Top-10 | R@20%E$_l$ | E@20%R$_l$ | IFA$_l$ |
|---|---|---|---|---|---|
| JITLine | 0.104 | 0.098 | 0.157 | 0.332 | 24.2 |
| Yan et al. | 0.193 | 0.195 | 0.143 | 0.345 | 15.3 |
| JIT-Fine | 0.212 | 0.214 | 0.208 | 0.318 | 10.8 |
| JIT-CF | **0.229** | **0.239** | **0.213** | **0.318** | **10.3** |

As shown in Table 6, JIT-CF consistently outperforms the baseline methods across all evaluation metrics, demonstrating its superior performance in defect localization. Notably, JIT-CF achieves the highest Top-5 accuracy of 0.229 and Top-10 accuracy of 0.239, which are higher than those of JIT-Fine, Yan et al. and JITLine. This indicates that JIT-CF is more effective in ranking the top defective lines within code commits.

JIT-CF yields the best performance in R@20%E$_l$ with a value of 0.213, and the lowest E@20%R$_l$ at 0.318. These results suggest that JIT-CF requires less effort to locate the same amount of defective lines compared to the other baselines, making it more efficient for developers in real-world scenarios. Additionally, the IFA$_l$ is also reduced to 10.3, indicating a significant decrease in false positives generated by the model, further enhancing its usability in practice.

### 6.2. What impact would more fully connected layers have on the model's performance?

In this section, we analyze the impact of increasing the depth of fully connected layers during the feature fusion process in JIT-CF. We extend our investigation to explore deeper architectures with up to 7 layers. Given that ReLU consistently achieved the best performance in RQ3, all experiments in this section are conducted using ReLU as the default activation function.

Our objective was to investigate whether increasing the number of fully connected layers could enhance the model's capability to capture complex patterns or, conversely, lead to performance degradation due to overfitting or computational inefficiencies. As shown in Fig. 5, we observe that while the performance of the model improved initially as we increased the layers from 1 to 2. However, any further increase in the number of layers led to either stagnation or a decline in performance.

Notably, the two-layer architecture emerged as the best performer across multiple metrics, achieving an optimal balance between model capacity and generalization. Adding more layers beyond two did not yield meaningful performance gains; instead, it often resulted in slight declines across metrics like F1-score, AUC, Recall@20%Effort, and Popt. For instance, while the F1-score peaked at 0.491 with two layers, it gradually decreased with more layers, with a marked drop at seven

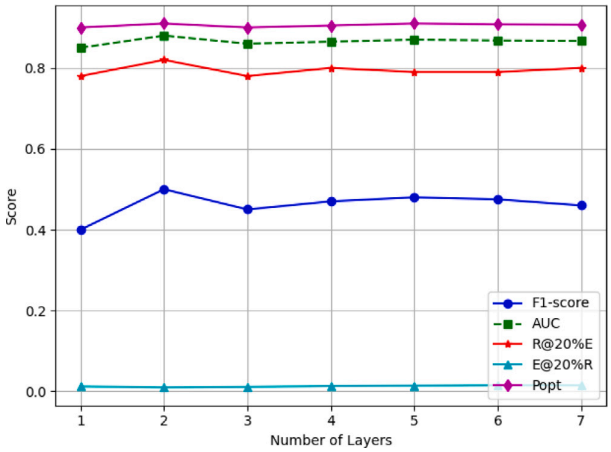



**Fig. 5.** Performance impact of 1 to 7 fully connected layers on JIT-CF.

**Table 7**
Performance comparison of JIT-CF using different pre-trained models.

| Model | F1-score | AUC | Recall@20%E | Effort@20%R | Popt |
|---|---|---|---|---|---|
| CodeBERT | **0.491** | **0.896** | **0.831** | **0.010** | **0.944** |
| GraphCodeBERT | 0.481 | 0.886 | 0.821 | 0.012 | 0.922 |
| CodeT5 | 0.483 | 0.876 | 0.811 | 0.011 | 0.912 |
| UniXcoder | 0.473 | 0.871 | 0.813 | 0.011 | 0.921 |

layers. Similarly, AUC and other metrics followed this trend, confirming that deeper architectures do not provide additional benefits in this context.

These results suggest that increasing the model depth beyond two layers introduces complexity that likely leads to overfitting, thereby reducing the model's ability to generalize to unseen examples. Furthermore, the marginal gains in some metrics do not justify the added computational cost, particularly when the overall performance begins to degrade as the model depth increases.

### 6.3. How do different pre-trained models impact JIT-CF's performance?

To explore the robustness and versatility of JIT-CF, we evaluated our framework using various pre-trained models, including CodeBERT, GraphCodeBERT, CodeT5, and UniXcoder [35,54–56]. The results are summarized in Table 7.

The results indicate that CodeBERT achieved the highest performance across all metrics, suggesting its effectiveness in capturing semantic and syntactic features for JIT defect prediction. GraphCodeBERT, which incorporates data flow information, performed slightly lower than CodeBERT but still demonstrated strong performance. This suggests that while data flow information is useful, the general semantic understanding provided by CodeBERT is more critical for defect prediction in this context. CodeT5, designed for code understanding and generation tasks, showed competitive results but was slightly less effective, likely due to its focus on broader code generation rather than specific defect detection. UniXcoder, which is optimized for cross-modal tasks, achieved the lowest performance among the tested models. This may be attributed to its training objectives, which are less aligned with the specific needs of defect prediction. Overall, these findings highlight the importance of selecting appropriate pre-trained models based on the specific requirements of the downstream task.

### 6.4. Threats to validity

In this subsection, we discuss potential threats to the validity of our research results and the measures taken to mitigate them.

**Threats to internal validity** primarily concern factors such as hyperparameter selection that could influence experimental outcomes. JIT-CF depends on hyperparameters such as learning rate, batch size, and the number of layers, all of which can significantly affect its performance. To mitigate this, we conducted comprehensive experiments to choose reasonable and stable parameter values based on standard practices and preliminary trials. While our main objective was to demonstrate the model's effectiveness rather than optimize every parameter, we believe that the chosen configurations fairly reflect the capabilities of JIT-CF.

**Threats to external validity** involve the generalizability of our results to other datasets and programming languages. To alleviate this threat, our evaluation was conducted on the JIT-Defects4J dataset, a widely adopted benchmark in JIT defect prediction research. Despite this, the dataset consists mainly of Java projects, which may limit the applicability of our findings to other languages and contexts. To address this, we made efforts to ensure that the design of JIT-CF remains flexible and applicable to other domains, although our experiments were focused on this specific dataset. When considering the application of JIT-CF to projects developed in other programming languages, several challenges may arise, including syntactic and semantic differences between languages, the availability and quality of pre-trained models, and language-specific coding practices. These factors may require adjustments to the semantic feature extraction process and the selection of expert features to better fit the context of different languages.

**Threats to construct validity** relate to the suitability of the metrics used to evaluate JIT-CF. To alleviate this threat, we employed a set of well-established metrics, including F1-score, AUC, Recall@20%E, Effort@20%R, and Popt, to provide a balanced assessment of the model's performance. Since these metrics are widely used in defect prediction literature, they provide a comprehensive view of the model's predictive and effort-aware capabilities. However, to mitigate potential issues with class imbalance in the dataset, we included effort-aware metrics like R@20%E and E@20%R to ensure the robustness of our evaluation.

## 7. Related work

### 7.1. Just-in-Time Defect Prediction

The field of Just-In-Time Defect Prediction has undergone significant evolution, starting from early work that relied on traditional handcrafted features to predict software defects. Kamei et al. [20] utilized features such as the number of modified lines, developer experience, and change history to identify potential defects. These traditional features became widely used in JIT-DP models. However, a debate emerged around the relative effectiveness of simple versus complex features. Studies like [57] showed that simpler models can often perform as well as, or even better than, complex ones, questioning the need for overly intricate features in JIT-DP.

As limitations of traditional feature-based models — such as their generalizability across projects — became apparent, deep learning approaches gained traction. Models like DeepJIT [23] and CC2Vec [25] leveraged semantic information directly from code changes. Unlike traditional models, these deep learning methods automatically learn feature representations, which enhances prediction performance and model robustness.

Recent advancements in JIT-DP have focused on integrating expert-defined features with semantic features to achieve better performance. For example, JIT-Fine [28] combines semantic features from CodeBERT with expert features, demonstrating significant improvements in defect prediction accuracy. Another notable work is MOJ-SDP [58], which models JIT-DP as a multi-objective optimization problem by defining two conflicting optimization goals. This approach leverages the complementary nature of expert metrics and semantic metrics

through model-level fusion, effectively combining these metrics using techniques like the maximum rule.

Different from previous studies, our proposed JIT-CF framework builds on these advancements by introducing contrastive learning to further enhance semantic feature representation. Unlike existing models such as MOJ-SDP and JIT-Fine, which rely solely on fusing expert and semantic metrics, JIT-CF leverages contrastive learning to maximize the similarity within positive pairs and minimize it between negative pairs. This approach significantly improves the model's ability to distinguish between similar code changes and enhances its robustness against noisy labels. Additionally, JIT-CF optimizes the feature fusion process through careful architecture design, specifically using a two-layer fully connected network with ReLU activation. This configuration allows the model to capture complex interactions between semantic and expert features, leading to superior performance.

### 7.2. Contrastive learning in pre-trained models

Contrastive learning has emerged as a powerful technique for learning robust representations by contrasting positive and negative samples [59,60]. Its application in pre-trained models has shown significant improvements across various tasks. The central idea is to bring semantically similar samples closer in feature space while pushing dissimilar samples further apart. This approach has been successfully integrated into pre-trained models like BERT and its variants [34,61].

Contrastive learning often uses data augmentation techniques to generate positive and negative pairs [62]. Methods like synonym replacement, back-translation, and random cropping of text segments are commonly employed to create diverse examples, enabling the model to learn more robust representations. By contrasting positive and negative pairs, the model captures fine-grained differences in data, leading to enhanced performance in downstream tasks.

In pre-trained language models, contrastive learning has been employed to improve the quality of learned representations. Researchers have explored supervised contrastive learning for fine-tuning BERT [61,63,64], yielding improvements in sentence-level classification tasks. Additionally, contrastive learning has also been applied in entity and relation extraction tasks, showcasing its versatility and effectiveness across different applications.

Despite its success in many NLP tasks, contrastive learning has seen limited application in JIT-DP. This study bridge the gap by integrating contrastive learning with a pre-trained model for JIT-DP, demonstrating significant performance improvements through enhanced feature representations and improved generalization.

## 8. Conclusion and future work

In this paper, we introduce JIT-CF, a framework for just-in-time defect prediction that leverages the advantages of contrastive learning and optimized feature fusion. By integrating contrastive learning with CodeBERT, JIT-CF enhances the semantic representations of code changes, thereby improving their discriminative capabilities. A key aspect of our approach is the architecture optimization during feature fusion, specifically through the selection of a two-layer fully connected network with ReLU activation, which is crucial for boosting the model's performance and robustness.

Our experiments on the JIT-Defects4J dataset demonstrate that JIT-CF significantly outperforms state-of-the-art baselines such as JIT-Fine and JITLine. The results confirm that the combination of contrastive learning for semantic feature extraction and architecture optimization in feature fusion effectively enhances JIT defect prediction performance. In future, we plan to apply JIT-CF to projects developed in other programming languages and explore more advanced contrastive learning methods to further enhance the model's capabilities. Additionally, we aim to investigate more sophisticated feature fusion techniques to optimize the model's overall performance and generalizability.

## CRediT authorship contribution statement

**Xiaolin Ju:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Conceptualization. **Yi Cao:** Writing – original draft, Visualization, Software, Methodology. **Xiang Chen:** Writing – review & editing, Supervision, Methodology. **Lina Gong:** Writing – review & editing, Supervision. **Vaskar Chakma:** Writing – review & editing, Visualization. **Xin Zhou:** Writing – review & editing, Validation, Software.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

The data that has been used is confidential.

## References

[1] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, Zhendong Su, Exploring and exploiting the correlations between bug-inducing and bug-fixing commits, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 326–337.
[2] Michael Zhivich, Robert K. Cunningham, The real cost of software errors, IEEE Secur. Priv. 7 (2) (2009) 87–90.
[3] Sunghun Kim, Hongyu Zhang, Rongxin Wu, Liang Gong, Dealing with noise in defect prediction, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 481–490.
[4] Xiang Chen, Dun Zhang, Yingquan Zhao, Zhanqi Cui, Chao Ni, Software defect number prediction: Unsupervised vs supervised methods, Inf. Softw. Technol. 106 (2019) 161–181.
[5] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, David Cok, Local vs. global models for effort estimation and defect prediction, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, IEEE, 2011, pp. 343–351.
[6] Chao Ni, Wang-Shu Liu, Xiang Chen, Qing Gu, Dao-Xu Chen, Qi-Guo Huang, A cluster based feature selection method for cross-project software defect prediction, J. Comput. Sci. Tech. 32 (2017) 1090–1107.
[7] Jaechang Nam, Sinno Jialin Pan, Sunghun Kim, Transfer defect learning, in: 2013 35th International Conference on Software Engineering, ICSE, IEEE, 2013, pp. 382–391.
[8] George G. Cabral, Leandro L. Minku, Emad Shihab, Suhaib 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, IEEE, 2019, pp. 666–676.
[9] Chao Ni, Xin Xia, David Lo, Xiang Chen, Qing Gu, Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction, IEEE Trans. Softw. Eng. 48 (3) (2020) 786–802.
[10] Xiang Chen, Yingquan Zhao, Qiuping Wang, Zhidan Yuan, MULTI: Multi-objective effort-aware just-in-time software defect prediction, Inf. Softw. Technol. 93 (2018) 1–13.
[11] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, Ahmed E. Hassan, A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes, IEEE Trans. Softw. Eng. 43 (7) (2016) 641–657.
[12] Wei Fu, Tim Menzies, Revisiting unsupervised learning for defect prediction, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 72–83.
[13] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, An empirical study of just-in-time defect prediction using cross-project models, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 172–181.
[14] Qiao Huang, Xin Xia, David Lo, Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2017, pp. 159–170.
[15] Shane McIntosh, Yasutaka Kamei, Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 560–560.
[16] Sunghun Kim, E. James Whitehead, Yi Zhang, Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2) (2008) 181–196.

[17] Chao Ni, Kaiwen Yang, Xin Xia, David Lo, Xiang Chen, Xiaohu Yang, Defect identification, categorization, and repair: Better together, 2022, arXiv preprint arXiv:2204.04856.

[18] Luca Pascarella, Fabio Palomba, Alberto Bacchelli, Fine-grained just-in-time defect prediction, J. Syst. Softw. 150 (2019) 22–36.

[19] Steven Young, Tamer Abdou, Ayse Bener, A replication study: just-in-time defect prediction with ensemble learning, in: Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, 2018, pp. 42–47.

[20] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, Naoyasu Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (6) (2012) 757–773.

[21] Wei Zheng, Tianren Shen, Xiang Chen, Peiran Deng, Interpretability application of the just-in-time software defect prediction model, J. Syst. Softw. 188 (2022) 111245.

[22] Lei Qiao, Yan Wang, Effort-aware and just-in-time defect prediction with neural network, PLoS One 14 (2) (2019) e0211359.

[23] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, Naoyasu Ubayashi, Deepjit: an end-to-end deep learning framework for just-in-time defect prediction, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories, MSR, IEEE, 2019, pp. 34–45.

[24] Xinli Yang, David Lo, Xin Xia, Yun Zhang, Jianling Sun, Deep learning for just-in-time defect prediction, in: 2015 IEEE International Conference on Software Quality, Reliability and Security, IEEE, 2015, pp. 17–26.

[25] Thong Hoang, Hong Jin Kang, David Lo, Julia Lawall, Cc2vec: Distributed representations of code changes, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 518–529.

[26] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, Lingming 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, 2021, pp. 427–438.

[27] Chao Ni, Xin Xia, David Lo, Xiaohu Yang, Ahmed E. Hassan, Just-in-time defect prediction on JavaScript projects: A replication study, ACM Trans. Softw. Eng. Methodol. ( TOSEM) 31 (4) (2022) 1–38.

[28] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, David 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, 2022, pp. 672–683.

[29] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, Charles Sutton, A survey of machine learning for big code and naturalness, ACM Comput. Surv. 51 (4) (2018) 1–37.

[30] Song Wang, Taiyue Liu, Lin Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 297–308.

[31] Sijie Mai, Ying Zeng, Shuangjia Zheng, Haifeng Hu, Hybrid contrastive learning of tri-modal representation for multimodal sentiment analysis, IEEE Trans. Affect. Comput. 14 (3) (2022) 2276–2289.

[32] Sijie Mai, Ying Zeng, Haifeng Hu, Learning from the global view: Supervised contrastive learning of multimodal representation, Inf. Fusion 100 (2023) 101920.

[33] Varsha Suresh, Desmond Ong, Not all negatives are equal: Label-aware contrastive loss for fine-grained text classification, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 4381–4394.

[34] T. Gao, X. Yao, Danqi Chen, SimCSE: Simple contrastive learning of sentence embeddings, in: EMNLP 2021-2021 Conference on Empirical Methods in Natural Language Processing, Proceedings, 2021.

[35] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al., CodeBERT: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics, EMNLP 2020, 2020, pp. 1536–1547.

[36] Chi Sun, Luyao Huang, Xipeng Qiu, Utilizing BERT for aspect-based sentiment analysis via constructing auxiliary sentence, in: Proceedings of NAACL-HLT, 2019, pp. 380–385.

[37] Chanathip Pornprasit, Chakkrit Kla 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, IEEE, 2021, pp. 369–379.

[38] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, Shanping Li, Just-in-time defect identification and localization: A two-phase framework, IEEE Trans. Softw. Eng. 48 (1) (2020) 82–101.

[39] Audris Mockus, David M. Weiss, Predicting risk of software changes, Bell Labs Tech. J. 5 (2) (2000) 169–180.

[40] Xinli Yang, David Lo, Xin Xia, Jianling Sun, TLEL: A two-layer ensemble learning approach for just-in-time defect prediction, Inf. Softw. Technol. 87 (2017) 206–220.

[41] Qiaozhi Liu, Xiaohua Gu, Qi Wang, Ziyuan Li, Just-in-time defect prediction: a survey, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2020, pp. 1390–1401.

[42] Ting Chen, Simon Kornblith, Mohammad Norouzi, Geoffrey Hinton, A simple framework for contrastive learning of visual representations, in: International Conference on Machine Learning, PMLR, 2020, pp. 1597–1607.

[43] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, Xiaohu Yang, Perceptions, expectations, and challenges in defect prediction, IEEE Trans. Softw. Eng. 46 (11) (2018) 1241–1266.

[44] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, Ross Girshick, Momentum contrast for unsupervised visual representation learning, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 9729–9738.

[45] Minseon Kim, Jihoon Tack, Sung Ju Hwang, Adversarial self-supervised contrastive learning, Adv. Neural Inf. Process. Syst. 33 (2020) 2983–2994.

[46] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al., Learning transferable visual models from natural language supervision, in: International Conference on Machine Learning, PMLR, 2021, pp. 8748–8763.

[47] Zonghan Yang, Yong Cheng, Yang Liu, Maosong Sun, Reducing word omission errors in neural machine translation: A contrastive learning approach, in: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019, pp. 6191–6196.

[48] Hongchao Fang, Sicheng Wang, Meng Zhou, Jiayuan Ding, Pengtao Xie, Cert: Contrastive self-supervised learning for language understanding, 2020, arXiv preprint arXiv:2005.12766.

[49] Dinghan Shen, Mingzhi Zheng, Yelong Shen, Yanru Qu, Weizhu Chen, A simple but tough-to-beat data augmentation approach for natural language understanding and generation, 2020, arXiv preprint arXiv:2009.13818.

[50] Nghi D.Q. Bui, Yijun Yu, Lingxiao Jiang, Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations, in: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2021, pp. 511–521.

[51] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, Claire Le Goues, Varclr: Variable semantic representation pre-training via contrastive learning, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2327–2339.

[52] Paras Jain, Ajay Jain, Contrastive code representation learning, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021.

[53] Erik Arisholm, Lionel C. Briand, Eivind B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, J. Syst. Softw. 83 (1) (2010) 2–17.

[54] Yue Wang, Weishi Wang, Shafiq Joty, Steven C.H. 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, 2021, pp. 8696–8708.

[55] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.

[56] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, Jian Yin, UniXcoder: Unified cross-modal pre-training for code representation, in: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2022, pp. 7212–7225.

[57] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Kenichi Matsumoto, The impact of automated parameter optimization on defect prediction models, IEEE Trans. Softw. Eng. 45 (7) (2018) 683–711.

[58] Xiang Chen, Hongling Xia, Wenlong Pei, Chao Ni, Ke Liu, Boosting multi-objective just-in-time software defect prediction by fusing expert metrics and semantic metrics, J. Syst. Softw. 206 (2023) 111853.

[59] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, Yang Liu, Contrabert: Enhancing code pre-trained models via contrastive learning, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 2476–2487.

[60] Xiao Wang, Guo-Jun Qi, Contrastive learning with stronger augmentations, IEEE Trans. Pattern Anal. Mach. Intell. 45 (5) (2022) 5549–5560.

[61] Yixuan Su, Fangyu Liu, Zaiqiao Meng, Tian Lan, Lei Shu, Ehsan Shareghi, Nigel Collier, TaCL: improving BERT pre-training with token-aware contrastive learning, in: North American Association for Computational Linguistics 2022, NAACL 2022, Association for Computational Linguistics (ACL), 2022, pp. 2497–2507.

[62] Shuai Chen, Jing Zhang, Ying Li, Qiaozhi Liu, Contrastive learning for bug detection with unsupervised data augmentation, in: Proceedings of the 44th International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 1246–1257.

[63] Taeuk Kim, Kang Min Yoo, Sang-goo Lee, Self-guided contrastive learning for BERT sentence representations, in: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2021, pp. 2528–2540.

[64] Yu Zhang, Hao Cheng, Zhihong Shen, Xiaodong Liu, Ye-Yi Wang, Jianfeng Gao, Pre-training multi-task contrastive learning models for scientific literature understanding, 2023, CoRR.