Advancing Software Development and Evolution through Multimodal Learning

Yanfu Yan

Xi'an, Shaanxi, China

Bachelor of Engineering, Xiamen University, China, 2016
Master of Science, University of Chinese Academy of Sciences, China, 2019

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

William & Mary
May, 2025

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Yanfu Yan

Approved by the Committee, May 2025

Committee Chair
Denys Poshyvanyk, Professor, Computer Science
William & Mary

Robert Michael Lewis, Associate Professor, Computer Science
William & Mary

Oscar Chaparro, Assistant Professor, Computer Science
William & Mary

Antonio Mastropaolo, Assistant Professor, Computer Science
William & Mary

Kevin Moran, Assistant Professor, Computer Science
University of Central Florida

ABSTRACT

Software systems play a vital role in modern society, supporting everything from mobile applications to enterprise solutions and critical infrastructure. As these systems continue to grow in size and complexity, software engineers face increasing challenges in maintaining their quality, reliability, and maintainability. Core tasks such as bug triaging, code search, and change impact analysis require a deep and comprehensive understanding of heterogeneous software artifacts, including source code, textual documentation, graphical user interfaces (GUIs), and structural dependencies. Traditional learning-based techniques, often limited to a single data modality, struggle to capture the rich, interconnected context necessary for accurate and scalable analysis in these domains.

This dissertation explores multimodal learning as a promising direction for addressing long-standing challenges in software engineering. By integrating multiple complementary data modalities, this research improves automation, accuracy, and robustness in critical software maintenance tasks. Specifically, it investigates the use of modern deep learning architectures, such as vision transformers, graph neural networks, and code-specific language models, in constructing multimodal representations of software artifacts. Three novel techniques are proposed, each demonstrating how multimodal integration can significantly improve the effectiveness of automated software engineering tools.

The first contribution, Janus, is a technique for detecting duplicate video-based bug reports, which are increasingly prevalent in GUI-centric mobile applications. Janus combines vision transformer-based scene understanding with sequential frame alignment to capture visual, textual, and sequential patterns on GUI screens, enabling nuanced comparison across video reports. Evaluation on an extended real-world benchmark demonstrates that Janus outperforms prior work by 9%, supported by both quantitative metrics and qualitative interpretability through hierarchical GUI representations.

The second contribution, Athena, addresses the critical task of change impact analysis. Traditional impact analysis techniques often rely on historical co-change data or dynamic execution traces, which can be brittle or expensive to collect. Athena instead introduces a hybrid framework that integrates conceptual coupling, derived from Transformer-based code embeddings, with structural dependencies encoded in program dependence graphs. This fusion enables more accurate and fine-grained impact predictions without the need for historical or run-time information. To support a reliable evaluation, we construct the first large-scale benchmark for impact analysis based on fine-grained untangled commit data, revealing that Athena achieves a performance gain of over 10% compared to state-of-the-art baselines.

The third contribution tackles the pressing issue of trustworthiness in deep code models, particularly in the face of out-of-distribution (OOD) inputs. As software models are deployed in open-world settings, they are increasingly exposed to inputs that differ from their training distribution, potentially leading to unreliable behavior. To address this, we propose COOD and COOD+, the first multimodal OOD detection frameworks tailored to code-related tasks. These techniques leverage contrastive learning and binary rejection mechanisms across both code and natural language comment modalities to detect anomalous inputs and recover performance degradation in critical downstream SE tasks such as code search.

Together, these contributions demonstrate the power of multimodal learning to overcome key limitations of traditional software engineering tools. By capturing the full complexity of real-world software artifacts, which span code, text, visuals, and structural dependencies, this research offers a unified and robust foundation for the next generation of intelligent software development tools. The proposed systems, models, and benchmarks not only improve automation and scalability, but also pave the way for more trustworthy, accurate, and context-aware solutions in software engineering.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

Above all, I would like to express my deepest appreciation to my advisor, Professor Denys Poshyvanyk, whose guidance, patience, and unwavering support have been instrumental throughout this journey. His vast knowledge, insightful feedback, and commitment to excellence have profoundly shaped my academic growth. I am especially grateful for his patience in challenging me to think critically, and for the countless hours he devoted to reviewing my work and offering constructive suggestions. Beyond his academic mentorship, his kindness and wisdom have been a constant source of encouragement in my personal life. I consider myself incredibly fortunate to have had him not only as an outstanding advisor but also as a role model for my future career. This dissertation would not have been possible without his invaluable support, and for that, I am deeply grateful.

I would also like to thank my Ph.D. committee members, Professor Robert Michael Lewis, Professor Oscar Chaparro, Professor Antonio Mastropaolo, and Professor Kevin Moran, for their valuable time, thoughtful feedback, and encouragement. Their insights have been essential in shaping the final version of this work.

My sincere thanks also go to my collaborators, Professor Kevin Moran, Professor Oscar Chaparro, Professor Gabriele Bavota, and many others. I am truly grateful for the opportunity to work with such exceptional researchers and for the guidance and ideas they have shared with me along the way.

Finally, I would like to thank my family and friends for their constant support and encouragement throughout this journey.

To my beloved mom and dad —
Your unwavering love, sacrifices, and guidance have shaped every step of my journey. Thank you for instilling in me the values of perseverance, optimistic, and compassion. This work is a reflection of your enduring support, and I dedicate it to you with all my heart.

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Software systems are essential to modern life, driving innovation and efficiency in areas ranging from mobile applications to enterprise solutions and critical infrastructure. However, as these systems grow in scale, functionality, and complexity, developers face increasing challenges to maintain their quality and reliability. Tasks such as understanding the cascade impacts of code changes, managing extensive repositories of diverse artifacts, and diagnosing complex issues require substantial effort and expertise. To address these challenges, researchers and practitioners have increasingly turned to advanced learning-based techniques to automate and optimize various stages of software development. Traditional approaches based on machine learning, which often rely on single data modalities [22,215], have proven useful but often fail to capture the global context required for sophisticated software engineering tasks. In fact, these tasks often span multiple modalities, requiring an integrated understanding of various forms of data. For example, a screenshot of the mobile app contains both visual demonstrations and textual information about the components, requiring tools capable of simultaneously processing and understanding both.

Software engineering is inherently multimodal, encompassing diverse artifacts that contribute to the development and maintenance of software systems. The key modalities include textual data, source code, visual data, and structural information, each offering distinct insights. Textual artifacts, such as bug reports, commit messages, and documenta-

tion, are ubiquitous in software engineering, providing context and descriptions of system behavior. The source code, a distinct modality, combines syntax, semantics, and structural relationships. Effectively representing and understanding code requires techniques that capture its meaning, structure, and context. The rise of graphical user interfaces (GUIs) has introduced visual artifacts such as screenshots [211], video-based bug reports [55], *etc.*, which often depict the visual behavior of software systems, providing insights that textual descriptions alone cannot convey. Structural information, characterized by dependencies and relationships between modules of software systems, plays a critical role in understanding interconnectivity. Represented through methods like data flow graphs and control flow graphs, structural data provide a global view of software module interactions. Together, these modalities enable for more comprehensive analysis and informed decision making in software development. Using their complementary strengths, multimodal learning produces richer representations of software artifacts, improving prediction accuracy and automation.

Despite its significant potential, multimodal learning in software engineering faces several challenges. The heterogeneity of the modalities is a primary concern, as each has distinct characteristics that require specialized processing techniques. Textual data are typically sequential, visual data are based on spatial relationships, and structural data is graph-based, necessitating tailored approaches for effective integration. Aligning information across modalities presents another complexity; for example, correlating local code semantics with global structural dependencies demands a nuanced understanding of how semantic and structural information interact. Scalability further complicates the application of multimodal learning, as software systems often involve large-scale data, including large volumes of code, numerous bug reports, and complex dependency graphs. Multimodal models must process this data efficiently without compromising performance. Lastly, software engineering tasks frequently require domain-specific expertise, such as proficiency in programming languages, knowledge of software design principles, and understanding of bug manifestation patterns. Integrating this expertise into multimodal

frameworks is crucial to ensuring their effectiveness in real-world scenarios.

Recent advances in deep learning have accelerated the adoption of multimodal approaches in software engineering. Transformer-based models such as CodeBERT [72] and GraphCodeBERT [81] have achieved strong performance by modeling data from both natural language (NL) and programming language (PL). For structural data, graph neural networks (GNNs) [84] are widely used to process program dependence graphs, enabling the integration of global structural information into localized representations. Meanwhile, vision transformers (ViTs) [62] have proven to be highly effective for processing visual data, such as screenshots and GUI components, capturing critical layout patterns for tasks such as GUI comprehension. These advances have collectively improved the ability to represent and integrate multimodal information, paving the way for more sophisticated solutions in software engineering.

This dissertation explores how multimodal learning can improve essential software engineering tasks, such as bug triaging and impact analysis. These tasks are chosen for their relevance and complexity as they represent core challenges in maintaining modern software systems. Bug triaging involves efficiently identifying, categorizing, and addressing software defects, while impact analysis predicts the cascading effects of code changes across a software system. Despite their importance, these tasks remain labor intensive, time consuming, and error-prone, particularly as software systems evolve. Addressing these challenges requires innovative approaches that integrate advanced computational techniques into software engineering workflows. To this end, this dissertation proposes several novel techniques, such as Janus and Athena, which exemplify how the combination of various data modalities can significantly improve automation and effectiveness in software development.

Specifically, Janus is designed to address the growing challenge of managing video-based bug reports, which are increasingly used to document issues in mobile and GUI-based applications. By combining visual representation learning, information retrieval and sequence-based algorithms, Janus analyzes the visual, textual, and sequential in-

formation present in video-based bug reports, improving the detection of duplicate bug reports. ATHENA, on the other hand, focuses on impact analysis (IA), enhancing traditional IA techniques by integrating conceptual coupling, derived from transformer-based neural models, with structural dependencies extracted from program dependence graphs. This hybrid approach enables more accurate and scalable impact predictions without relying on historical commits or execution traces.

Beyond improving performance on these core tasks, this dissertation addresses a broader concern in modern ML-driven SE: the trustworthiness of deep code models. Specifically, we examine the problem of out-of-distribution (OOD) detection, which is essential in open-world settings where models encounter data that deviates from their training distribution. Failing to recognize such input can lead to unreliable or erroneous predictions. Motivated by the gaps in current SE research on OOD detection, we introduce COOD and COOD+, the first multimodal OOD detection frameworks tailored for code-related tasks. These techniques enhance the robustness of a key downstream task, namely code search, by accurately identifying and managing OOD inputs.

In general, this dissertation contributes novel methods, systems, and evaluation benchmarks that advance the field of multimodal learning in software engineering. By integrating heterogeneous sources of information, including text, code, visuals, and structural relationships, these contributions demonstrate how SE tools can become more intelligent, trustworthy, and more aligned with the complexities of real-world software development.

1.1 Task Focus and Contributions

1.1.1 Detecting Duplicate Video-based Bug Reports through Multimodal GUI Scene Learning

Video-based bug reports have become a promising alternative to text-based reports for programs centered around a graphical user interface (GUI), as they allow for seamless documentation of software faults by visually capturing buggy behavior on app screens.

However, developing automated techniques to manage video-based reports is challenging, as it requires identifying and understanding often nuanced visual patterns that capture key information about a reported bug.

To this end, we propose a new approach JANUS, which adapts the scene learning capabilities of vision transformers to capture subtle visual and textual patterns that manifest on GUI screens of the application - which is key to differentiate between similar screens for accurate duplicate detection. Our approach also makes use of a video alignment technique capable of adaptive weighting of video frames to account for sequential bug manifestation patterns. We created a comprehensive benchmark for evaluation (the largest benchmark to date) by drastically extending a prior dataset with real bugs as opposed to injected bugs from prior work. The evaluation results demonstrate that our approach outperforms previous work by 9%, with statistical significance. Additionally, we qualitatively illustrate how the improved performance JANUS's benefits from its scene-learning capabilities through interpretable hierarchical GUI representations.

1.1.2 Code Change Impact Analysis via Multimodal Coupling

Impact analysis (IA) is a critical software maintenance task that identifies the effects of a given set of code changes on a larger software project with the intention of avoiding potential adverse effects. IA is a cognitively challenging task that involves reasoning about the abstract relationships between various code constructs. Given its difficulty, researchers have worked to automate IA with approaches that primarily use coupling metrics as a measure of "connectedness" of different parts of a software project. Many of these coupling metrics rely on static, dynamic, or evolutionary information and are based on heuristics that tend to be brittle, require expensive execution analysis, or large histories of co-changes to accurately estimate impact sets.

To address these challenges, we introduce ATHENA, a novel approach that first combines graph information from the (structural) dependence of a software system with a conceptual coupling approach that uses advances in deep representation learning for code

without the need for change histories and execution information. The code embeddings are augmented through a graph-based propagation strategy to integrate global dependence information into local code semantics. Prior benchmarks are small and suffer from tangled commits, making it difficult to measure accurate results. Therefore, we built the first large-scale impact analysis benchmark based on fine-grained commit information from bug fixes. On this more reliable benchmark, ATHENA shows a significant improvement of 10% over the baselines. In collaboration with Cisco Systems [1], this work has revitalized research on impact analysis, a fundamental SE task that has been largely overlooked since 2018.

1.1.3 Towards More Trustworthy Deep Code Models by Enabling Multimodal Out-of-Distribution Detection

When developers utilize state-of-the-art deep code models for various understanding and generation tasks (e.g., code search, conceptual coupling used in our ATHENA approach), it is crucial to ensure that the predictions produced by these models are trustworthy. However, these models are typically developed under the assumption that training and testing data come from the same distribution, which is often violated in the open world since deployed models may frequently encounter OOD instances (OODs) that are not seen in the training. Hence, when confronted with OODs, a reliable and trustworthy code model must be capable of detecting them to either abstain from making predictions or potentially forward these OODs to appropriate models handling other distributions or tasks. Although OOD detection has been studied in the machine learning field, it has not yet been explored in the SE context for code-related tasks.

Pre-trained natural language Transformers have been shown to be vulnerable to OODs, so we hypothesize that pre-trained Transformer-based code models are similarly susceptible. To verify this, we systematically investigate the ability of pre-trained code models to detect OODs and the impact of OOD detection on a downstream code task (*i.e.,* code search). Our findings reveal that the performance of a code search model drops by around 5% due to the presence of OODs. Therefore, we developed the first multi-modal (*i.e.,* com-

ment and code) OOD detection framework for pre-trained code models using contrastive learning in both unsupervised (COOD) and weakly-supervised settings (COOD+) [7]. As distribution shifts can occur in either modality (comment or code) or both, COOD+ integrates an improved contrastive learning module with a binary OOD rejections module and devises a new scoring metric to fuse their prediction results. For evaluation, we constructed the first OOD benchmark tailored for the code context with multiple (four) scenarios. Extensive experimental results demonstrate the effectiveness of our COOD and the integration of two modules in COOD+ for detecting OODs from different scenarios in two modalities. More importantly, the performance loss of the code search model caused by the presence of OODs is recovered by utilizing our COOD/COOD+ detector.

1.2 Organization

The remainder of this dissertation is organized as follows. Chapter 2 reviews the background relevant to this work, focusing on existing learning-based techniques for deriving representations of software artifacts to support key tasks such as bug triaging and change impact analysis. Chapter 3 presents the design and implementation of Janus, a system to detect duplicate video-based bug reports in GUI-intensive mobile applications. It describes how Janus leverages vision transformers and adaptive video alignment to model multimodal patterns, and reports the evaluation results on an extended benchmark. Chapter 4 introduces Athena, a novel framework for the analysis of change impact that integrates deep code representation learning with structural dependency information to predict the cascading effects of code changes. This chapter also details the creation of a new large-scale benchmark and demonstrates Athena's improvements over state-of-the-art approaches. Chapter 5 turns to the challenge of ensuring the reliability of deep-code models in real-world software engineering environments. It presents COOD and COOD+, the first multimodal out-of-distribution (OOD) detection frameworks designed for code-related tasks. These systems integrate contrastive learning and binary OOD rejection across code and

natural language modalities to identify anomalous inputs and improve the robustness of downstream tasks in open-world settings. Finally, Chapter 6 concludes the dissertation by summarizing the key contributions of this work and reaffirming the value of multimodal learning in enabling more intelligent, trustworthy, and scalable software analysis tools.

1.3 Bibliography Notes

This proposal is based on the author's previously published work, and permission has been obtained from the publishers and all co-authors to reprint sections of it:

- **Y. Yan**, V. Duong, H. Shao, and D. Poshyvanyk, "Towards More Trustworthy Deep Code Models by Enabling Out-of-Distribution Detection," *in Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, 13 pages.
- **Y. Yan**, N. Cooper, K. Moran, G. Bavota, D. Poshyvanyk, and S. Rich, "Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs," *in Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*, 2024, 24 pages.
- **Y. Yan**, N. Cooper, O. Chaparro, K. Moran, and D. Poshyvanyk, "Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports," *in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, 13 pages.
- **Y. Yan**, "On Improving Management of Duplicate Video-based Bug Reports," *in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2024, 3 pages.
- A. Chen, **Y. Yan**, and D. Poshyvanyk, "ACER: An AST-based Call Graph Generator Framework," in Proceedings of the IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), 2023, 6 pages.

Chapter 2

Background

This chapter provides the necessary background for understanding the learning-based techniques central to the dissertation proposal. We begin by discussing methods for visual representation learning, highlighting both convolutional and Transformer-based approaches. Next, we examine optical character recognition (OCR) techniques used to extract textual information from GUI screens. Finally, we explore advances in code representation learning, focusing on both traditional and deep learning-based methods that enable semantic understanding of software artifacts.

2.1 Visual Representation Learning

CNN-based Techniques. Visual Representation learning has become a popular field in the Computer Vision domain [131]. The purpose of this research area is to learn high-quality visual representations that are helpful for downstream tasks such as image classification [135], object detection [250], or image captioning [101]. Generally, this visual representation task is performed unsupervised, self-supervised, or supervised [50, 176]. Most recently, a focus has been on contrastive methods [50, 176] and distillation methods [42]. While there has been a wealth of work on visual representation learning, it is important to understand the techniques employed by past work on duplicate detection for

video-based bug reports. The most closely related work to JANUS is the TANGO approach introduced by Cooper *et al.* [55], where the authors used a self-supervised contrastive method called SimCLR [50]. This technique uses data augmentation to generate different views of a given image by training a Siamese network based on the temperature-scaled cross entropy loss in order to learn representations that are invariant to image changes. However, there are notable downsides of this approach including: (i) the need for large collections of negative examples to facilitate the contrastive loss function, and (ii) limitations related to the traditional Convolutional Neural Networks (ConvNets), upon which SimCLR is built, which can have difficulty in distilling subtle visual patterns from the learned representation that are essential for video de-duplication.

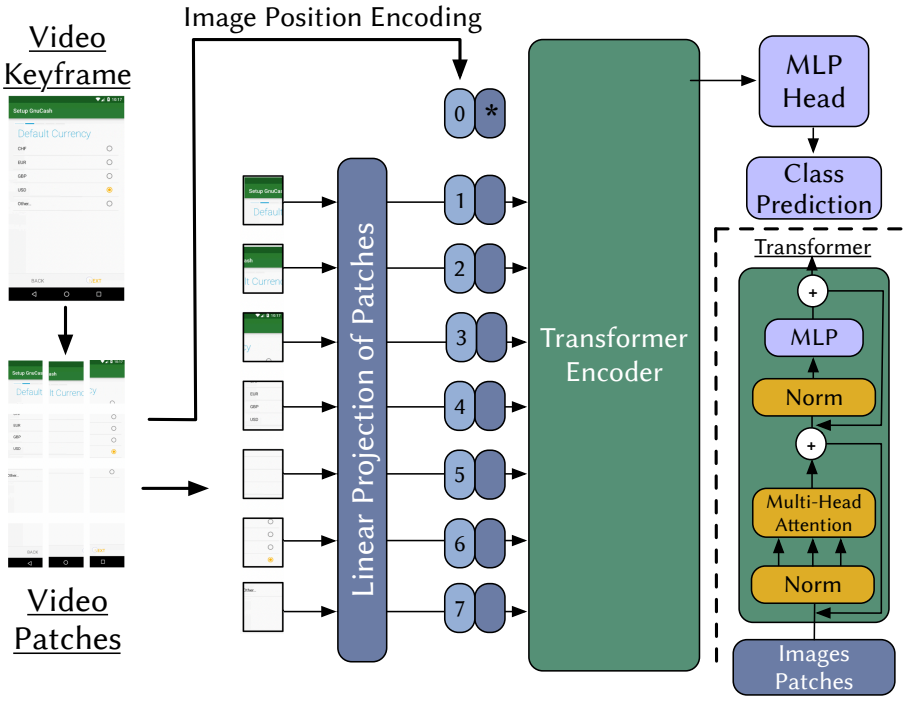

Figure 2.1: The Vision Transformer (ViT) Architecture.

Transformer-based Techniques Recent advancements in deep learning from visual data have, similar to other fields such as natural language processing, embraced the Transformer architecture [62] to learn visual representations. A popular recent approach, called DINO [42], is capable of learning explicit information about the semantic segmentation

of an image (*i.e.,* object boundaries), which is likely a direct consequence of adopting the Vision Transformer (ViT) architecture [62] and a self-supervised learning procedure. We posit that learning object segmentation within an image will be particularly useful for images of app GUIs, given their structured, component-based nature. An overview of the ViT architecture is shown in Figure 2.1. As illustrated, it is made up of a standard Transformer encoder model [60] but instead of lexical tokens, "patches" of the images are fed into the network. These image patches are treated the same way that tokens are treated in lexical transformers: they are linearly transformed and have added positional embeddings. Given that image-level supervision requires laborious annotations and limits the information that can be learned during training to a single concept with a few categories of objects, Caron *et al.* [42] proposed a self-supervised training methodology for DINO, which utilizes a student-teacher knowledge distillation training scheme [100].

2.2 Optical Character Recognition

Optical Character Recognition (OCR) is the process of converting images of typed, printed, or handwritten text into machine-encoded text through text localization and recognition. Text localization identifies textual regions, *i.e.,* it distinguishes text from the background by using object detection models that are capable of localizing text bounding boxes at the word or text line level. The content in these regions is then recognized and converted into natural language tokens based on an encoder-decoder architecture [198]. Historically, OCR techniques have combined learning-based techniques with a number of heuristics to aid in pruning false positives or merging neighboring image regions, making technique pipelines quite complicated [196]. In prior work on Tango, the authors used the popular Tesseract OCR [2] library, which segments components of the screen likely to contain text by generating binary maps of the images to first obtain image regions and then uses the long short term memory (LSTM) [78] model to recognize text on them.

2.3 Code Representations Learning

To generate vectorized code representations, information retrieval (IR) approaches (*e.g.,* LSI, term frequency - reverse document frequency (TF-IDF), latent dirichlet allocation (LDA)) were first used to support SE tasks. They typically require building a corpus from all documents (code artifacts) and then representing code by measuring the importance of each code token to a document in the corpus and/or exploiting co-occurrences of code tokens based on singular value decomposition (SVD) or Bayesian topic modeling. However, these IR approaches treat the code as *bag-of-words*, ignoring the order and semantics of code tokens. Thus, neural networks have been used to obtain more meaningful code representations. For example, word2vec [163] takes into account each individual token and its context tokens by using a sliding context window during training. Furthermore, doc2vec [138] could learn a paragraph vector for variable length code, instead of using an average representation of code tokens as word2vec does. Subsequently, more and more end-to-end deep models (*e.g.,* Bi-RNN [52], TextCNN [125], Self-Attention [209]) have been used to extract code embeddings.

Representation learning refers to the process of learning a parametric mapping from raw input data to a low-dimensional latent space with the goal of extracting more abstract and useful features that enhance performance across various downstream tasks. In recent years, self-supervised representation learning has gained significant traction in the machine learning community, largely due to the success of large pretrained models. The pretraining-then-finetuning paradigm [31, 60, 177, 237] has proven particularly effective in natural language processing (NLP), where models are first pretrained on massive text corpora in a self-supervised manner to learn general-purpose representations, and then fine-tuned on task-specific datasets. The Transformer [209] architecture stands out as the most representative encoder backbone for this scheme. With the advent of large-scale code datasets (*i.e.,* CodeSearchNet [107]), this scheme has also been increasingly applied to learn code representations and auto-build software engineering tasks [17, 80, 216, 218].

CodeBERT [72] was one of the first transformer-based pre-trained models to support various code-related tasks. It distinguishes between the programming language (PL) and the natural language (NL) modality and captures their semantic connection during pretraining. However, it only utilizes the sequential information of the bi-modal data while ignoring the inherent structure of code. Therefore, GraphCodeBERT [81] further incorporates data flow information within methods into sequenced code snippets during pre-training, resulting in enhanced code embeddings. Other pretrained models like UniXcoder [80] that encode abstract syntax tree (AST) information to produce syntax-aware code embeddings.

2.3.1 Multimodal Contrastive Learning

Contrastive learning [85] is an emerging technique that learns discriminative representations from data organized into similar / dissimilar pairs. It has been developed over multiple subfields, such as computer vision (CV), NLP, [50, 122], and SE [80, 108]. Recently, researchers have developed multimodal approaches that combine contrastive learning with multiple data modalities, achieving superior performance over unimodal models in various tasks [150, 176]. Due to the prowess of contrastive learning in discriminative tasks (*e.g.,* classification and information retrieval), it naturally fits the OOD detection domain, as shown by studies on unimodal and multimodal data [67,165,175]. Inspired by this, we applied contrastive learning to NL-PL data to learn representative and discriminate features to train our OOD detector.

Chapter 3

Detecting Duplicate Video-based Bug Reports through Multimodal GUI Scene Learning

Video-based bug reports are increasingly being used to document bugs for programs centered on a graphical user interface (GUI). However, developing automated techniques to manage video-based reports is challenging, as it requires identifying and understanding often nuanced visual patterns that capture key information about a reported bug. In this project, our goal is to overcome these challenges by advancing the bug report management task of *duplicate detection* for video-based reports. To this end, we introduce a new approach, called Janus, which adapts the scene-learning capabilities of vision transformers to capture subtle visual and textual patterns that manifest on the application UI screens – which is key to differentiate between similar screens for accurate detection of duplicate reports. Janus also makes use of a video alignment technique capable of adaptive weighting of video frames to account for typical bug manifestation patterns. In a comprehensive evaluation on a benchmark containing 7,290 duplicate detection tasks derived from 270 video-based bug reports from 90 Android app bugs, the best configuration of our approach

achieves an overall mRR/mAP of 89.8%/84.7%, and for the large majority of duplicate detection tasks, outperforms prior work by ≈9% to a statistically significant degree. Finally, we qualitatively illustrate how the scene learning capabilities provided by JANUS benefit its performance.

3.1 Introduction

Video-based bug reports are becoming increasingly popular for mobile applications [48, 70, 71, 137]. As mobile app bugs are typically manifested visually via the graphical user interface (GUI), recording videos depicting bugs is more natural compared to textual bug reports [43, 48, 69, 71, 197]. App users can easily record app bugs through the recording features of mobile operating systems (*e.g.,* Android [6]) or through third-party recording apps [5]. In addition, popular issue trackers, such as GitHub [11], offer easy-to-use features for users to submit these videos to app developers. Recent studies have documented the rapidly increasing use of videos in mobile app issue trackers [70, 137]. Feng *et al.* studied open source apps hosted on FDroid [9] and reported the usage of more than 13k video recordings on issue trackers between 2012 and 2020, with a significant increase in usage during 2018-2020 (*i.e.,* a 15% - 35% increase). Kuramoto *et al.* [137] reported a 13% increase in videos-containing issues in 2017-2021 for 289k popular GitHub projects.

Although video-based bug reporting offers various advantages (easiness of recording and submission, and visual details about app bugs [48, 55, 70, 71, 137]), it also presents several challenges for developers during bug report management tasks, particularly in scenarios where a high volume of bug reports is encountered [48, 70, 71, 137].

One of the most challenging tasks for developers is determining whether video-based bug reports depict the same app bug. This situation arises when multiple users independently report identical problems with the application (*e.g.,* during crowd-sourced app testing [55, 63, 88, 148]). In such scenarios, developers face the challenge of watching, understanding and assessing incoming and previously submitted video-based bug reports.

This task can be extremely challenging, as these recordings typically show numerous steps executed quickly, making it difficult to recognize the bug-reproduction scenario from the videos [48,55,71]. Additionally, the behavior of buggy apps may not be apparent in videos of the various types of bugs that apps can show on their GUI [66]. Developers often need to pause and replay the videos multiple times in order to fully understand the reported problems [48, 71]. The task of detecting duplicate (video-based) bug reports is crucial during the bug triage process, as it helps developers avoid excessive redundant effort to investigate and resolve identical problems [55, 63, 88, 211].

This challenge is particularly prominent in the crowd-sourced testing of mobile apps [63, 88], wherein software vendors engage a large distributed user base to test applications in various operational environments, for example, encompassing various devices, locations and mobile networks. Crowd-sourced app testing often leads to multiple users encountering and reporting the same app-related issues. In fact, previous research has found that a substantial proportion (80%+) of bug reports submitted by users during crowd-sourced app testing are duplicates [211]. Consequently, developers often spend considerable effort on duplicate detection, which can impede the overall bug resolution process [55,63,88,211].

This dissertation proposes JANUS, a novel automated approach designed to assist developers in identifying duplicate video-based bug reports. JANUS combines visual representation learning, information retrieval, and sequence-based algorithms to analyze visual, textual, and sequential information present in video-based bug reports. Through these methods, JANUS establishes the degree of similarity between videos when reporting the same bug, enabling the automated detection of duplicate reports.

To model visual information within videos, JANUS leverages the Vision Transformer (ViT) architecture [62] and the DINO self-supervised training scheme DINO [42], which extract rich hierarchical features that explicitly capture scene layout information related to GUI screens. In addition, JANUS analyzes the textual content of videos using the Efficient and Accurate Scene Text Detector (EAST) [253] and a Transformer-based Optical Character Recognition (TrOCR) model [143], which accurately localize and extract

text from video frames. By encoding this textual content via an adapted vector space model (VSM) [77], Janus assesses the textual similarity between two videos. Finally, to encode the sequential aspect of videos, Janus incorporates an adapted version of the classical longest common substring algorithm, giving higher weight to subsequent video frames that show the behavior of the buggy app even if the videos show different bug reproduction scenarios.

We evaluated Janus using a comprehensive benchmark of 7,290 duplicate detection tasks, constructed from 270 video-based bug reports that represent 90 unique bugs found in nine Android apps. We created this benchmark by extending an existing data set that relied mainly on synthetic bugs [55]. Specifically, we extended it by incorporating 90 video-based bug reports pertaining to 30 real bugs of different kinds (*e.g.,* crashes, incorrect app output, and cosmetic issues) from three additional apps, resulting in a more comprehensive, realistic, and diverse benchmark.

Through multiple ablation experiments, we systematically assess the performance of the individual components of Janus and various combinations of these components. Our evaluation demonstrates that the most optimal configuration of Janus (when visual, textual, and sequential video information is combined) achieves an overall mRR / mAP of 89.8% / 84.7%, exceeding the performance of an existing duplicate detector by ≈9% (with statistical significance). These results suggest that Janus can significantly reduce the effort required to identify duplicate video-based bug reports, as developers would only need to review fewer video reports to assess whether an incoming report depicts a known bug.

Furthermore, we conducted a qualitative analysis to understand the reasons behind Janus' performance compared to prior work. In particular, Janus exhibits an interpretable representation of video frames, effectively capturing nuanced patterns related to the style, composition, and layout of the GUI component, which are crucial in accurately distinguishing duplicate video-based bug reports.

3.2 Related Work

GUI Comprehension. Understanding GUI can help with many software engineering tasks related to mobile applications, such as GUI reverse engineering [24, 46, 166, 248], software testing [25,155,167,174,241], and GUI search [26,45,47]. Most GUI understanding techniques need to detect GUI elements first to understand the information provided by the GUI. Chen *et al.* [49] show that deep learning-based object detection models [64, 179, 180] and scene text detector EAST [253] outperform old-fashioned detection models [169] and the OCR tool Tesseract [195], respectively. Fu *et al.* [75] utilize the Transformer architecture for GUI element detection, but only based on limited pixel words. The most closely related work to our own [55] uses the self-supervised approach SimCLR [50] based on ResNet [90] to understand the visual GUI and use OCR to obtain textual information in order to detect duplicate video-based bug reports.

Duplicate Video Retrieval. To retrieve similar videos, traditional techniques in the computer vision domain first extract global and/or local features of video frames, then aggregate extracted features to represent a whole video, and finally calculate similarity scores between videos. Visual features are extracted by hand-crafted image processing methods, such as Local Binary Patterns (LBP) [113, 227], Scale-Invariant Feature Transform (SIFT) [225, 249], or Convolutional Neural Networks (CNN) [90, 203]. The features can then be aggregated based on global vectors [225], bag-of-words [53, 132], or deep metric learning [133]. Kordopatis-Zilos *et al.* [134] conducted a comprehensive experimental study comparing feature extraction methods, CNN architectures, and aggregation schemes, showing that CNN+BoVW is the best performing combination, which is the reason why the most relevant work [55] chose this strategy to obtain video representations.

3.3 The JANUS Duplicate Detector

This section describes the architecture and design details behind JANUS, our approach to detect duplicate video-based bug reports.

3.3.1 Problem Formulation and Challenges

We formulate the problem of duplicate detection as an information retrieval (IR) problem, as is typical for textual bug reports [120, 147, 173, 246]. A newly submitted video-based bug report (the query) is compared to the set of previously submitted video reports in the issue tracker (the corpus) using a retrieval engine (*e.g.,* JANUS), which retrieves and ranks the corpus reports according to their similarity to the query. The higher a video-based report is ranked, the more likely it is to depict the same bug as the query. A developer would then watch the ranked videos in a top-down manner, marking the new video as duplicate if they find a video depicting the same bug. Video-based bug reports depict the incorrect behavior of an application (*e.g.,* GUI screens showing a crash, layout problems, or functional misbehavior), and the actions performed by the user on the GUI screens that lead to such misbehavior (*i.e.,* steps to reproduce the bug). Duplicate video-based bug reports are pairs of two reports, *e.g.,* the query report and one corpus report, that depict the same buggy behavior, possibly showing different GUI steps, as multiple sequences of steps can lead to the manifestation of the same buggy behavior. An advantage of an IR formulation over other methods (binary classification *e.g.,* as (non-)duplicate reports [44]) is the fact that a ranked list gives higher flexibility to developers, because multiple bug reports are recommended as possibly showing the same bug.

Although the primary goal of a duplicate detector is to identify whether two distinct videos depict the same incorrect app behavior, there are multiple challenges that make this task particularly difficult. For example, duplicate videos may vary in length and display different reproduction steps, originating from various reproduction scenarios executed by users or the omission of certain steps during recording. Even if the reproduction steps appear to be the same or highly similar in all videos, users can execute them at varying speeds. Distinguishing between different videos that display distinct yet similar unexpected app behavior and reproduction steps can pose challenges to detectors. Furthermore, certain applications may display dynamic content. For example, a mobile web

browser allows users to navigate websites with varying layouts and content.

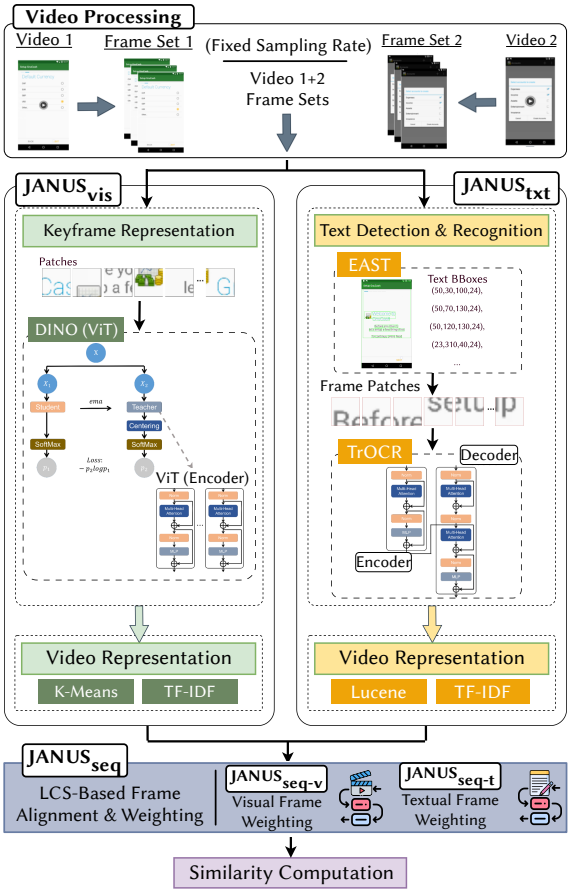

Figure 3.1: Overview of the JANUS duplicate detector.

3.3.2 Janus Overview

An overview of JANUS is shown in Fig. 3.1. JANUS receives as input two video-based bug reports and outputs a similarity score that indicates how similar they are to depict the same app bug. JANUS can be used to compute the scores between a new video-based bug report and a corpus of videos that represent previously submitted bug reports. The scores allow for ranking the corpus videos as a list of potential duplicate candidates. The goal of JANUS is to rank higher in this list the actual duplicates for the new video.

Internally, JANUS begins sampling a number of frames from the two videos at a given

rate (every sixth frame following the findings of previous work [55]) to reduce overhead, since successive frames tend to be exact or near duplicates of each other. Next, JANUS computes a vector representation of the videos by processing the visual and/or textual content of the frames. JANUS's visual component, JANUS$_{vis}$, vectorizes each video into a visual TF-IDF representation by discretizing the frames into a Bag of Visual Words (BoVW) [110], using a feature extractor based on a Vision Transformer (ViT) model [62] and the DINO self-supervised training scheme [42]. JANUS's textual component, JANUS$_{txt}$, vectorizes each video into a textual TF-IDF representation by extracting the text from the video frame (via the EAST [253] and TrOCR [143] models) and constructing a document of the concatenated text, represented as a Bag of Words (BoW) [187]. Each pair of visual or textual TF-IDF representations is then compared via cosine similarity. Visual and textual similarities can be used individually to rank duplicate candidates, or combined into a single similarity score to account for both modalities of information, ideally leading to more effective duplicate detection.

To account for the sequential nature of video-based bug reports, which typically show the reproduction steps first and the incorrect app behavior afterward, JANUS can compute an alternative similarity score, based on a customized version of the longest common substring (LCS) algorithm, which matches the vector representation of video frames via cosine similarity and produces an overall similarity score that weights the later frames in the video more heavily than the earlier ones. This similarity is calculated by the sequential component JANUS's, JANUS$_{seq}$, which operates on the visual (JANUS$_{seq-v}$) and textual (JANUS$_{seq-t}$) vector representations of the frames.

3.3.3 JANUS$_{vis}$: Visual Representation of Videos

JANUS$_{vis}$ obtains a visual representation of a video in two steps. First, the sampled video frames are resized to 224×224 (*pixels*) and encoded using visual representation learning [131]. Second, these frame embeddings are further processed into a Bag of Visual Words (BoVW) [134], which is used to represent a video as a TF-IDF vector [187]. The

goal is to learn useful visual information from the GUI components and layouts of the app shown in the videos to distinguish potential duplicates from non-duplicates.

3.3.3.1 Visual Representation of Video Frames

Visual representation learning aims to obtain high-quality visual representations that are helpful for downstream tasks such as image classification [135], object detection [250], or image captioning [101]. This task is typically carried out unsupervised, self-supervised, or supervised [50, 176]. Most recently, a focus has been on contrastive [50, 176] and distillation learning methods [42]. A promising technique, known as the Vision Transformer (ViT) [62] has recently been proposed to better understand visual representations. The performance of this architecture has been demonstrated to surpass or, at the very least, match previous models relying on Convolutional Neural Networks (CNNs) for image classification. However, the most significant advantage of ViT lies in its ability to excel beyond CNNs in capturing explicit information concerning the semantic segmentation of an image (*i.e.,* layouts and object boundaries) [62].

We posit that learning object segmentation within an image is particularly useful for app GUI screens, given their structured, component-based nature. Hence, we adopted the ViT architecture for designing JANUS$_{vis}$. The ViT architecture is made up of a standard Transformer encoder model [60] but instead of lexical tokens, "patches" of the images are fed into the network. These patches are treated in the same way that tokens are treated in lexical transformers: they are linearly transformed and have added positional embeddings.

Given that image-level supervision requires labor-intensive annotations and limits the information that can be learned during training to a single concept with a few categories of objects (as is the case of app GUI screens, which contain components and layouts of well-defined kinds), we need to train our ViT model in a self-supervised manner. JANUS$_{vis}$ trains its ViT using the DINO self-supervised training methodology DINO [42], which leverages a student-teacher knowledge distillation training scheme [100]. In this scheme, The student network is trained to match the distribution of the teacher network by min-

imizing the standard cross-entropy loss. Usually, the teacher network is larger than the student network in terms of the number of model parameters. However, the teacher network in DINO is built from the past iterations of the student network with an exponential moving average strategy, whose parameters are frozen over an epoch by applying a stop-gradient operator, given that direct replication of the student weights fails to converge. The outputs of both networks are normalized using a softmax temperature. To adapt the knowledge distillation architecture to self-supervised learning, two global views and several local views are constructed on the basis of data augmentations [79] and the multi-crop strategy [41], with local views passed through the student while only the global views are passed through the teacher network, to encourage local-to-global correspondence. By combining DINO with ViT, we aim to further improve the ability to capture global GUI layouts.

Through this self-supervised training process, the model learns a rich representation of images that emphasize scene layouts and object boundaries. To further refine the DINO model's capabilities to our domain of app GUI screens, we fine-tuned the JANUS's ViT model, which was pre-trained on ImageNet [59], on a collection of 66k mobile app screenshots from the popular RICO dataset [58]. We directly use the projected output of the *[CLS]* token, a special token that marks the weighted aggregation of all image patch embeddings, from the last block of the ViT model as the representation of video frames.

3.3.3.2 Visual Representation of Videos

To represent a video, JANUS$_{vis}$ implements a BoVW + TF-IDF approach, since it has been shown to be more useful for video retrieval compared to other approaches [134] (*e.g.,* using directly the frame representations for similarity computation or aggregating them into a single vector).

JANUS$_{vis}$ discretizes the frame representations by leveraging a Codebook of visual words [134]. The Codebook represents a catalog of visual words, which are representative vectors found in a corpus of images (in our case, images of app GUIs). The Codebook is

constructed using a trained K-Means model that clusters the corpus of image representations into K clusters, the centroids being the visual words. JANUS$_{vis}$ then assigns each video frame representation to its closest cluster centroid (*i.e.,* a visual word) via Euclidean distance. The Codebook is trained by randomly sampling 15k mobile app screenshots from the RICO dataset [58], vectorizing them via our fine-tuned ViT model, and running the K-Means algorithm on the vectors, with $K = 1k$ recommended by prior work [134]. We take a sample rather than using the entire RICO dataset due to computational constraints of the K-Means algorithm. The codebook is trained only once before the TF-IDF representation approach is applied.

Once each frame representation is discretized to its corresponding visual word, JANUS$_{vis}$ computes a TF-IDF vector representation of a video, as was similarly done for text retrieval [187]. The term frequency (TF) is the count of each visual word in the video. The inverse document frequency (IDF) is the count of BoVW representations of existing videos where a visual word appears. Since a corpus of existing videos for a particular app may be small and may lack diversity, we consider the set of RICO images as the corpus of existing videos. By considering the diversity of apps in the RICO dataset, we aim to improve the generalization of the TF-IDF video representations.

JANUS$_{vis}$ compares the TF-IDF representation of two videos via cosine similarity to establish the likelihood that the videos show the same app bug. This method is applied to the existing corpus of TF-IDF visual representations for an app to generate a ranked list of candidate duplicate videos for a new video-based bug report.

To address potential biases due to random sampling when creating the codebook, we adapted JANUS$_{vis}$ to use four codebooks (each trained on 15k RICO images, 60k in total). Specifically, JANUS$_{vis}$ uses each codebook to produce similarity scores for a set of videos. These similarity scores are averaged to produce a final set of similarities and video ranking. More details are given in section 3.4.3.2.

3.3.4 JANUS$_{txt}$: Textual Representation of Videos

JANUS$_{txt}$ creates a textual representation of a video in two steps: (1) it localizes and extracts the text present in the video frames via neural text localization and Optical Character Recognition (OCR); and (2), it encodes the extracted text using a standard TF-IDF representation [187]. The goal is to take advantage of text from labels, messages, and other sources shown in the frames to compute video similarity.

For the first step, JANUS$_{txt}$ has two components: (1) a text localization component that proposes image regions where text is rendered, and (2) a text recognition component that takes those regions and extracts any text present in them. The text localization component implements the Efficient and Accurate Scene Text Detector (EAST) model [253], which has been trained to directly derive region proposals. The text recognition component uses the TrOCR Transformer model [143], which takes proposals from the EAST region and directly predicts the text represented in the proposals. The combination of EAST and TrOCR was adopted over the popular TesseractOCR [2] approach because: (1) such a combination simplifies the overall OCR pipeline since it relies only on neural models, without needing heuristic-based approaches to filter out poor text region candidates (as TesseractOCR does); and (2) such a combination has shown strong performance improvements in detecting scene text as well as handwritten/printed text, which means it is less sensitive to noise in the images. Each video frame is put through this 2-stage pipeline to extract its text.

For the second step, JANUS$_{txt}$ concatenates the text from all video frames and preprocesses it by tokenization, lemmatization, and removal of special characters, such as non-ASCII characters, punctuation, or stop words. This resulting text is used to build a Bag of Words (BoW) representation of the video, which is then encoded as a standard textual TF-IDF representation using the popular Lucene library [77], which implements the standard Boolean information retrieval model and the Vector Space Model (VSM) [187]. We use this textual representation approach over neural text encoding models because it

is based on exact text matching, which could lead to more accurate similarity computation of duplicate videos (as they are likely to show the same text on the buggy app screens).

Finally, JANUS$_{txt}$ compares the TF-IDF representation of two videos using Lucene's similarity scoring function (based on cosine similarity and document length normalization) [14]. The similarity computation can be applied to a corpus of video-based bug reports to generate a ranked list of possible duplicate videos to the new video.

3.3.5 JANUS$_{seq}$: Sequential Similarity of Videos

JANUS$_{vis}$ and JANUS$_{txt}$ ignore the sequential order of the videos, as these components are based on Bags of (Visual) Words. However, the buggy app behavior is typically shown toward the end of a video-based bug report, after the bug reproduction steps have been rendered. To account for the sequential order of the videos, JANUS employs a modified version of the longest common substring algorithm (LCS) to calculate an alternative similarity score between videos. This approach is coined as JANUS$_{seq}$ and operates on both visual (JANUS$_{seq-v}$) and textual representations of videos (JANUS$_{seq-t}$).

JANUS$_{seq}$ treats a video as a sequence of visual/textual words, based on the vector representation of the video frames, and applies an LCS-based approach for similarity computation. Intuitively, the longer the LCS between videos, the higher their similarity. The textual representation of a video frame is the TF-IDF vector of the text extracted from the frame, using the approach described in section 3.3.4. In the standard word-based LCS algorithm, words are compared using exact text matching. To account for similar, yet different video words (which might be common for textual video representations), we relaxed this matching scheme and instead used cosine similarity between video frame representations. Additionally, similarity-based matching should weigh more heavily the frames that appear later in the videos, as they are more likely to show the buggy app behavior and should give a normalized similarity score between zero and one.

Given these requirements, we defined the following similarity computation for JANUS$_{seq}$: $S_{seq} = \frac{\text{w-}LCS}{\max \text{ w-}LCS}$, where the numerator, w-LCS, represents the amount of overlap between

two videos, given by our modified LCS algorithm, which uses the cosine similarity between frames (rather than exact matching) and a weighting scheme that favors later frames in the videos. The weighting scheme is $\frac{i}{m} \times \frac{j}{n}$, where i is the ith frame of a first video, with m being its # of frames, and j is the jth frame of a second video, with n being its # of frames. The denominator, max w-LCS, represents the maximum possible overlap if the videos were identical. Since the videos could be of different lengths, we align the end of the shorter video (with length min), to the end of the longer video (with length max), and calculate the maximum overlap as: $\sum_{i=1}^{min} \frac{i}{min} \times \frac{max-i}{max}$.

3.3.6 Combining Janus's Components

To design JANUS, we explore different combinations of its components. The similarity scores of JANUS$_{vis}$ and JANUS$_{txt}$ can be linearly combined as $(1-w) \times S_{vis} + w \times S_{txt}$, with $w \in [0, 1]$— the higher w is, the more weight it gives to the textual information of the videos. We also explore various combinations that replace this similarity calculation with those given by JANUS$_{seq}$ (JANUS$_{seq-v}$ & JANUS$_{seq-t}$), which consider sequential video information.

3.4 Evaluation Methodology

We investigate the performance of JANUS's components (JANUS$_{vis}$, JANUS$_{txt}$, and JANUS$_{seq}$), as well as the performance of various combinations of these components, and compare these to a baseline duplicate detection technique proposed in prior work [55]. Furthermore, our objective is to understand why we observe various trends in the overall performance of JANUS and qualitatively examine the cases where JANUS is able to outperform the baseline technique. To that end, we formulate the following research questions (RQs):

RQ$_1$: *What is* JANUS$_{vis}$'S *duplicate detection performance?*

RQ$_2$: *What is* JANUS$_{txt}$'S *duplicate detection performance?*

RQ$_3$: *What is* JANUS$_{seq}$'S *duplicate detection performance?*

RQ$_4$: *What is the performance of* JANUS*'s component combinations?*

3.4.1 Duplicate Detection Dataset

We constructed a comprehensive evaluation dataset by extending a prior dataset that mainly relied on synthetic app bugs [55]. The previous data set collected 60 distinct bugs (35 crashes and 25 non-crashes) in six Android apps of different sizes and domains (*e.g.,* podcast, finance and weight management apps). The data set contains ten confirmed real bugs and 50 bugs injected by the MutAPK mutation testing tool [66], which generates code mutations based on diverse mutant operators that affect various features of the app. The data set includes three duplicate videos per bug, for a total of 180 video-based bug reports, and a set of 810 duplicate detection tasks per app, for a total of 4,860 tasks created from the videos. We refer to this dataset as the *original dataset*. Next, we describe how we extended this dataset and detail the creation of video-based bug reports and duplicate detection tasks to evaluate JANUS.

3.4.1.1 Extended Real Bug Dataset

We extended the previous data set by building an evaluation data set containing *only* real bugs. Wendland *et al.* [220] released the AndroR2 data set containing 90 manually reproduced bug reports for Android apps. This data set was then extended by adding more 90 reproduced bug reports in the AndroR2+ dataset [114], for a total of 180 real and reproducible reports. For each bug report, AndroR2+ provides a link to the original bug report in the issue tracker, an `apk` of the version of the buggy app, a reproduction script and metadata for bug reproduction (device, OS version, *etc.*).

To construct our new real bug dataset, we chose the three apps with the largest number of bugs from AndroR2+, while also ensuring the diversity of app categories. We selected: Firefox Focus (FCS) [10], a web browser; PDF Converter (ITP) [13], an image-to-PDF

converter; and GPSTest (GPS) [12], a GPS testing app. FCS is the only app that renders dynamic content on the screen. For these apps in AndroR2+, we found ten bug reports for FCS, nine reports for GPS, and eight reports for ITP. We further manually checked each app's issue tracker and collected one more bug for GPS and two more bugs for ITP to have the same number of bugs per app. To find the `apkfiles` of the correct buggy version of the apps for these three bugs, we chose the version of the app closest to the date the issue was created and confirmed that the `apk` allowed a successful reproduction of the bug. Based on the AndroR2+ metadata and the three bug reports we collected, there are seven different OS versions used to reproduce the bugs, namely, Android version 4.4.4, 6.0.1, 7, 7.1, 8, 8.1, & 9.

3.4.1.2 Duplicate Video Recording

The authors of the article and the external participants recorded videos replicating the 30 real bugs collected from the three AndroR2 + apps, following previous work [55].

We re-wrote the descriptions of the steps to reproduce (S2R), expected behaviors, and observed behaviors for these bugs to ensure that they are clear and easy for participants to reproduce from an end-user perspective. Although AndroR2 + bugs were reproducible in a Pixel 2 emulator, we chose Nexus 5X to maintain the same device configuration as the previous dataset [55], since bugs were also reproducible in Nexus 5X. This ensures a consistent resolution of the videos across the benchmarks. Additionally, we minimized the different OS versions to three (6, 8.1, and 9) to reduce the participants' effort by finding the closest OS versions to their original ones while ensuring the bugs were still reproducible. Also, having these additional OSes in our video reproductions of these bugs has the added benefit of being more realistic—the prior dataset only used Android 7.0. Although AndroR2 + provides automated bug reproduction scripts, we avoided using them for two reasons: (i) we found that certain scripts led to errors that did not properly reproduce the bug and (ii) we wanted to capture video-based reports depicting real human actions, to ensure the most realistic setting possible.

The authors of the paper created video reports for the 30 bugs according to the S2R. To maintain three duplicate videos per bug, in line with the previous dataset, two authors (who previously did not record any videos) along with two Ph.D. students were asked to record the additional 60 videos, each responsible for reproducing 15 distinct bugs with only the descriptions of expected and observed behaviors, to ensure diversity of reproduction steps. Unlike the previous data set, the recorded videos do not show the Android touch indicator when the user taps the screen.

In total, our new dataset consists of 90 video-based bug reports corresponding with three duplicates of 30 real bugs from three apps. It contains two crashes and 28 non-crashes, comprising 270 reproduction steps in total (249 taps, six gestures, and 15 input entry actions) and ≈35-second videos, on average. There are six videos for Android 6, nine for Android 9, and 75 for Android 8.1.

3.4.1.3 Duplicate Detection Tasks

In line with the previous data set, we construct *duplicate detection tasks* for each app to be as realistic as possible. We define a duplicate detection task as having: (1) a *query* video that represents a newly reported video-based bug report, and (2) a *corpus* of 13 existing video-based reports. The query must be compared against the corpus in order to determine whether the incoming report is a duplicate of an existing report. Each task contains videos of the 10 bugs for an application. The corpus contains two duplicate videos of the query (*i.e.,* they show the same bug). The remaining 11 videos are non-duplicates: three of them are duplicates of each other but not of the query (*i.e.,* they show a bug different from the query bug), and eight videos show distinct bugs. Each task simulates a situation that is similar to crowd-sourced app testing, where duplicates of the query, of other bugs, and unique video-based reports exist together on the issue tracker for an app.

Using different combinations of bugs and videos, we created a total of 810 tasks per app or 2,430 tasks across all apps. Combining both the previous and the new datasets, there are 7,920 tasks in our extended evaluation benchmark to evaluate Janus.

3.4.2 Baseline Duplicate Detector

We compare JANUS against the TANGO duplicate detector introduced by Cooper *et al.* [55]. TANGO also leverages multimodal information to detect duplicate video-based reports, using less sophisticated methods compared to JANUS. It extracts visual features from video frames using a contrastive learning method called SimCLR, which uses a ResNet-50 CNN to learn local features of app GUIs [50]. It also analyzes text displayed on GUI screens using an approach that combines LSTM-based language models and heuristics, relying on TesseracOCR to extract text from video frames [2]. Finally, TANGO performs a limited alignment of video frames: only for its visual SimCLR features extracted. TANGO's evaluation found the best performing configuration is when the visual and textual components are combined, hence we compare JANUS against this configuration while also performing ablation comparisons between their individual components.

3.4.3 Metrics and Experimental Settings

3.4.3.1 Evaluation Metrics

We use standard metrics used in previous work on duplicate bug report detection evaluations [55, 120, 147, 246]:

- **Mean Reciprocal Rank (mRR)**: gives a measure of the average ranking of the first duplicate video found in the candidate list of videos given by a duplicate detector. It is calculated as: $mRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$, for N duplicate detection tasks ($rank_i$ is the rank of the first duplicate video for task i).

- **Mean Average Precision (mAP)**: it gives a measure of the average ranking of all duplicate videos for a query video. It is calculated as: $mAP = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{DV}\sum_{v=1}^{DV}P_i(rank_v)$, where DV is the set of duplicate videos for task i, $rank_v$ is the rank of the duplicate video v, and $P_i(k) = \frac{duplicates}{k}$ is the number of duplicates in the top-k candidates.

All metrics give a normalized score in [0, 1]—the higher the score, the higher the duplicate detection performance. We executed different configurations of JANUS and the baseline on the 7,920 tasks and computed/compared the metrics between these approaches.

3.4.3.2 Model Configurations

We compared JANUS$_{vis}$ against TANGO's visual component by experimenting with two ViT models: ViT-Small (ViT-S) and ViT-Base (ViT-B), which have six and 12 self-attention heads, respectively. ViT-S has a size similar to RestNet-50's size (used by TANGO's SimCLR): ≈23M parameters. To evaluate the differences between SimCLR (contrastive) and DINO (distillation) training schemes, we implemented JANUS$_{vis}$ with DINO + RestNet-50. We also experimented with the following patch sizes for ViT: 16×16 (/16) and 8×8 (/8) pixels, as patch size can affect JANUS$_{vis}$'s performance [62]. In total, we executed four DINO models: DINO (ResNet), DINO (ViT-S/16), DINO (ViT-S/8), & DINO (ViT-B/16). ViT-B/8 was not included in the experimentation for JANUS$_{vis}$ due to its substantial computation overhead.

To account for potential biases from random image selection when constructing the JANUS$_{vis}$'s Codebook, we used four distinct Codebooks, each trained on 15k distinct RICO images (60K images in total). With each codebook, JANUS$_{vis}$ generates similarity scores for a set of videos. These similarities are averaged across the four codebooks to produce final scores used for ranking. To perform a fair comparison with the visual component of TANGO, we implemented the same codebook generation strategy in TANGO, using its publicly released implementation [55]. The recomputed results TANGO in the previous dataset are slightly higher than those reported in the original paper (76.2 vs. 75.3 mRR and 69.8 vs 67.8 mAP).

We compared JANUS$_{txt}$ against TANGO's textual component by experimenting with different configurations for the EAST and TrOCR models. For EAST, we used three different resolution thresholds to filter out small text regions: 5×5 (EAST-5), 40×20 (EAST-40), and 80×40 (EAST-80). The 5×5 threshold is used by default in EAST. We

Table 3.1: The network configurations and fine-tuning hyperparameters for JANUS$_{vis}$ compared with SimCLR used by TANGO

model	dim	# params	batch size	w-temp	temp
SimCLR	2,048	23M	1,792	–	–
DINO (ResNet)	2,048	23M	96	0.03	0.03 (0)
DINO (ViT-S/16)	384	21M	96	0.03	0.03 (0)
DINO (ViT-S/8)	384	21M	18	0.04	0.05 (30)
DINO (ViT-B/16)	768	85M	64	0.05	0.07 (50)

did not test larger resolutions than 80×40 to ensure that each textual document created for the video has at least one valid detection. 40×20 was included as a middle ground to understand the impact of threshold size on video similarity calculation. For TrOCR, we used its large version with BEiT Large [23] as the encoder and RoBERTa Large [153] as the decoder. Two fine-tuned TrOCR-Large models are used, namely TrOCR-p (fine-tuned on the printed text dataset SROIE [106]) and TrOCR-s (finetuned on the synthetic scene text datasets such as ICDAR15 [121] and SVT [212]).

3.4.3.3 Model Training

All visual models were fine-tuned in the 66k mobile app screenshots from the RICO dataset [58] for 100 epochs using model checkpoints trained on ImageNet [59], except for DINO (ViT-B/16), to fairly compare it with the TANGO$_{vis}$'s SimCLR model. After examining preliminary results showing the advantages of DINO with ViT, we decided to train DINO (ViT-B/16) for 400 epochs [42]. Fine-tuning was performed on three NVIDIA T4 Tesla GPUs with 16GB of memory each. Because DINO does not use contrastive learning, we were able to use a much smaller batch size compared to the SimCLR model used in TANGO: 96 *vs* 1,792 for ViT-S/16 and ResNet-50. For the ViT-B/16 and ViT-S/8 models, we used a batch size of 64 and 16 due to memory constraints. Table 3.1 shows the network configurations and three hyperparameters to fine-tune, where *dim* is the representation dimension of the output, *# parameters* is the total number of model parameters. *"temp" and "w-temp"* represent the teacher temperature and the warm-up teacher temperature, respectively, and the numbers in parentheses are the # epochs used

for warm-up. Model training was not required for JANUS$_{txt}$ as we directly use pre-trained EAST and TrCOR models for GUI text localization and recognition [143, 253].

3.5 Evaluation Results

Table 3.2 shows JANUS's duplicate detection performance compared to the baseline TANGO, for their individual components: visual, textual, and sequential. Table 3.3 shows the performance of different combinations of components JANUS, compared to the baseline.

Cells shaded green in these tables indicate a statistically significant (via Wilcoxon's paired test at the $p < 0.05$ level) higher effectiveness when comparing a given JANUS configuration/component to a given TANGO configuration/component. Yellow-shaded cells indicate higher performance, but without statistical significance. We present the results for each app of the *original* (mostly synthetic bugs) and *extended* (real bugs) *datasets* and the general results that account for all the apps in both sets, separately and together.

While we calculated the performance of four JANUS$_{vis}$ DINO models (*i.e.,* DINO with ResNet, ViT-S/16, ViT-S/8 and ViT-B/16), we present (in tables 3.2 and 3.3) the best performing model for JANUS$_{vis}$: DINO with ViT-B. Likewise, we report here the results of the best performing model configuration for JANUS$_{txt}$, namely EAST-80 (EAST that filters out region proposals smaller than 80×40) combined with TrOCR-s (TrOCR fine-tuned on real-world scenes, *e.g.,* street scenes, instead of text found in printed and handwritten documents). The results for all DINO, EAST, and TrOCR configurations can be found in our replication package [231].

Tables 3.2 and 3.3 show a consistent trend: the performance achieved by any duplicate detector (*i.e.,* any configuration) is lower for the original dataset than for the extended dataset. After investigating the minimal set of ground truth reproduction steps of bugs used in the datasets, we found that this trend is explained by the number of overlapping steps between distinct bugs in an app. We observed that distinct bugs for a given app in the original dataset have a larger step overlap than distinct bugs in the extended

dataset. It is more challenging for a duplicate detector to distinguish between duplicate and nonduplicate videos if there is a larger step overlap across bugs (hence, across videos). Recall that in a duplicate detection task, the videos in the corpus are for distinct bugs; if there is a larger overlap among them, particularly between duplicates and nonduplicates, a detector would struggle to discern the differences.

3.5.1 RQ1: JANUS$_{vis}$'s Performance

Table 3.2: Performance of the individual components of JANUS and the baseline TANGO

App	Visual				Textual				Sequential (visual)				Seq. (textual)	
	Tango		JANUS$_{vis}$		Tango		JANUS$_{txt}$		Tango		JANUS$_{seq-v}$		JANUS$_{seq-t}$	
	mRR	**mAP**	**mRR**	**mAP**	**mRR**	**mAP**	**mRR**	**mAP**	**mRR**	**mAP**	**mRR**	**mAP**	**mRR**	**mAP**
APOD	77.19	69.98	87.32	79.79	80.80	75.30	79.76	73.09	55.01	44.85	84.45	71.11	73.40	68.09
DROID	68.43	58.82	80.77	71.44	67.90	64.70	78.88	72.52	46.54	37.91	61.49	50.88	76.19	73.64
GNU	81.53	75.83	81.83	75.54	84.50	82.30	89.53	81.28	55.91	43.37	71.41	58.82	52.79	43.81
GROW	83.53	78.60	87.46	80.33	76.80	69.00	82.65	77.38	74.57	64.46	92.14	84.57	77.45	76.24
TIME	70.26	65.35	73.76	69.46	47.40	37.70	64.80	56.67	50.85	43.62	63.14	56.18	69.34	66.16
TOK	76.03	70.37	81.11	71.33	61.30	53.30	53.95	44.48	38.13	33.36	53.39	43.22	54.00	47.83
Original	**76.16**	**69.83**	**82.04**	**74.65**	**69.80**	**63.70**	**74.93**	**67.57**	**53.50**	**44.59**	**71.00**	**60.80**	**67.20**	**62.63**
FCS	91.09	85.82	86.88	82.69	85.12	79.12	86.17	84.88	65.23	55.42	90.20	85.91	90.53	88.21
GPS	95.99	92.15	98.09	95.70	92.11	84.82	97.51	96.10	68.34	60.63	93.72	86.64	57.83	53.33
ITP	81.93	73.92	93.29	84.08	89.73	86.34	96.77	89.83	69.50	54.37	90.56	78.20	54.74	46.87
Extended	**89.67**	**83.96**	**92.75**	**87.49**	**88.98**	**85.74**	**93.48**	**90.27**	**67.69**	**56.81**	**91.50**	**83.58**	**67.70**	**62.80**
Overall	**80.66**	**74.54**	**85.61**	**78.93**	**76.14**	**70.06**	**81.11**	**75.14**	**58.23**	**48.67**	**77.84**	**68.39**	**67.36**	**62.69**

Table 3.2 shows the duplicate detection effectiveness of JANUS$_{vis}$ (DINO with ViT-B) compared to visual TANGO (SimCLR).

Before discussing the table results, we briefly discuss the results of comparing the training schemes (distillation via DINO *vs.* contrastive via SimCLR, both using the same pre-trained ResNet weights). We found that SimCLR outperforms DINO for six of nine apps by a relatively small margin (by 3.5% mRR and 4.2% mAP, on average), but DINO outperforms SimCLR for the remaining three apps (APOD, GNU, DINO) by a larger margin (7.5% mRR and 6% mAP, on avg.). Overall, across all the apps, we found a similar performance between these two approaches (less than 1.1% mRR/mAP improvement), which indicates the training scheme does not have a large impact on duplicate detection performance.

Furthermore, both ViT-S/16 and ViT-S/8 used by JANUS$_{vis}$'s DINO exhibit superior performance compared to ResNet-50 used by visual TANGO's SimCLR. Specifically, al-

though ViT-S/16 and ViT-S/8 have a similar model size to RestNet-50, they outperform ResNet-50 by 2.91% and 2.92% respectively, in terms of mRR on average, with statistical significance. This highlights the effectiveness of ViT over ResNet for duplicate video-based bug report detection.

Table 3.2 shows that JANUS$_{vis}$ (DINO with ViT-B) significantly outperforms the baseline in both datasets. We observe an overall improvement of (85.61 - 80.66)/80.66 = 6.1% mRR and (78.93 - 74.54)/74.54 = 5.9% mAP, with statistical significance. This overall performance results from an improvement in eight of nine apps compared to the baseline (seven with statistical significance), with TANGO only having a substantial improvement over JANUS$_{vis}$ for the FCS app. These FCS results are due to the nature of the app and the underlying models. Specifically, FCS is a web browser and the video-based bug reports produced for this app show users navigating to different websites. The app produces dynamic content: the navigated websites have different layouts and visual characteristics. JANUS$_{vis}$'S ViT is prone to focusing more on the structure of the GUIs, extracting global features about the layouts, while the baseline's ResNet tends to focus on local visual features of the GUIs, not necessarily on general screen layouts, which are more beneficial to detect duplicates. Compared to ResNet, ViT's emphasis on GUI layouts leads to a more substantial dissimilarity between duplicates when sequential visual information is not taken into account.

3.5.2 RQ2: JANUS$_{txt}$'S Performance

Table 3.2 shows that JANUS$_{txt}$ is substantially more effective than textual TANGO for seven of nine apps (with statistical significance), especially for DROID (improvement of 16.2% mRR and 12.1% mAP) and TIME (improvement of 36.7% mRR and 50.3% mAP). Only for the apps APOD and TOK, TANGO is higher, resulting in JANUS$_{txt}$'s overall superiority on both the original and the real bug datasets (overall, by 6.5%/7.3% mRR/mAP). The reason why JANUS$_{txt}$ does not perform better for APOD and TOK is that these apps usually contain short or small pieces of text (*e.g.,* due to small fonts) on many of their

screens, and EAST fails to identify them because these pieces fit in smaller regions than 80×40 pixels. Indeed, when reducing the threshold to 40×20, JANUS$_{txt}$ outperforms TANGO for APOD and TOK (by 1.5%/1.1% and 8.8%/5.2% mRR/mAP respectively).

JANUS$_{txt}$'S performance is slightly higher than JANUS$_{vis}$'S for the extended dataset (improvement of 0.8%/3.1% mRR/mAP overall), but lower for the original dataset (by 9.5%/10.5% mRR/mAP). The lower improvements come from the TOK app, which does not contain enough textual information to accurately detect duplicates [55].

3.5.3 RQ3: JANUS$_{seq}$'S Performance

Table 3.2 shows that JANUS$_{seq-v}$ is substantially more effective in detecting duplicates than sequential TANGO, when using visual frame representations. JANUS$_{seq-v}$ outperforms the baseline for every app in the original dataset (by 32.7% mRR and 36.4% mAP overall) and in the extended dataset (by 35.2% mRR and 47.1% mAP overall). The high improvements can be attributed to the power of JANUS's ViT in learning the global structure of GUI screens, while TANGO's ResNet focuses on learning local GUI features. Global GUI structure representations are more useful to measure the sequential overlap between video frames even when there are small variations in the frames because of slightly different reproduction steps. Also, we note that the improvements for the FCS app are substantial (38%/55% mRR/mAP). Since we observed very different GUI layouts in video frames for this app (because users navigated to different websites), these results are indicative of the effectiveness of the sequential similarity approach of JANUS in combination with ViT-based frame representations (compared to the baseline).

Since TANGO's sequential component is not designed to work with textual frame representations (unlike JANUS), we only compare the performance of JANUS$_{seq-t}$ with JANUS$_{seq-v}$. In general, JANUS$_{seq-v}$ outperforms JANUS$_{seq-t}$ by 15.6%/9.1% mRR/mAP. It substantially outperforms JANUS$_{seq-t}$ for five of nine apps by 39.4%/ 35.8% mRR/mAP on average, while having a lower performance for the remaining four apps (7.4%/14.6% mRR/mAP). The largest improvement is observed in the GPS and ITP apps. ITP is an

app used to convert images to PDF, involving mostly image editing, while GPS focuses on editing coordinates and displaying locations on a map. Consequently, video-based bug reports have limited text on each frame, which negatively impacts the performance of JANUS$_{seq-t}$. However, since JANUS$_{txt}$ leads to high performance for these two applications, we attribute JANUS$_{seq-t}$'s relatively low performance to the alignment approach, which processes each video frame text rather than using the text from all frames together.

Table 3.3: Performance of different component combinations for JANUS and the baseline TANGO

App	Visual + Textual				Vis + Seq		Txt + Seq		Vis+Txt+Seq	
	Tango		Janus		Janus		Janus		Janus	
	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP
APOD	81.08	75.11	86.72	80.64	86.59	77.30	91.54	86.30	94.95	86.55
DROID	71.74	64.31	83.95	78.98	75.50	64.96	89.26	83.15	88.56	81.06
GNU	89.16	85.92	89.62	81.75	83.48	74.18	84.24	73.80	90.58	81.58
GROW	86.61	80.73	89.84	86.32	91.40	86.88	83.56	80.96	93.32	90.72
TIME	65.06	59.23	67.51	64.31	71.33	63.68	73.69	69.67	74.88	71.92
TOK	71.11	63.95	75.51	62.59	63.35	57.30	55.23	48.57	75.92	67.91
Orig.	**77.46**	**71.54**	**82.19**	**75.76**	**78.61**	**70.72**	**79.59**	**73.74**	**86.37**	**79.95**
FCS	91.11	86.87	88.46	85.95	93.98	89.85	90.38	84.74	94.73	91.90
GPS	97.35	95.53	99.30	98.31	98.09	96.01	89.20	86.91	98.24	97.26
ITP	90.64	86.51	96.84	91.58	96.05	89.99	83.94	77.07	97.41	93.51
Ext.	**93.03**	**89.64**	**94.87**	**91.94**	**96.04**	**91.95**	**87.84**	**82.91**	**96.79**	**94.22**
Overall	**82.65**	**77.57**	**86.42**	**81.16**	**84.42**	**77.79**	**82.34**	**76.80**	**89.84**	**84.71**

3.5.4 RQ4: Component Combination Performance

We linearly combined the JANUS's components (as described in section 3.3.6) to determine how much they improve performance, compared to the baseline and individual components. We experimented with different weights (from 0 to 1 in 0.1 increments) using all duplicate detection tasks and selected the weights that lead to the highest mRR/mAP performance.

As mentioned earlier, the best TANGO configuration is when its visual and textual components are combined (with a weight of 0.8 and 0.2, respectively), as reported in the original paper [55]. JANUS's visual and textual components (*i.e.,* JANUS$_{vis}$ and JANUS$_{txt}$) are combined using 0.9 and 0.1 as weights. This combination is denoted as "Visual +

Textual" in Table 3.3. The table also shows the combination of JANUS's visual/textual components and the sequential one: "Vis + Seq" denotes the average of the similarity scores produced by JANUS_{vis} and JANUS_{seq-v}, while "Txt + Seq" denotes the average of the similarity scores produced by JANUS_{txt} and JANUS_{seq-t}. An average combination means a weight of 0.5. Finally, we combine the similarities produced by the last two combinations using a weighted linear combination as follows: Sim(Vis + Seq) × 0.6 + Sim(Txt + Seq) × 0.4. This combination incorporates every information source from the videos and is denoted as "Vis + Txt + Seq".

Table 3.3 shows that the best performing JANUS combinations are "Visual + Textual" and "Vis + Txt + Seq", both outperforming the baseline by 4.6%/4.6% mRR/mAP and 8.7%/9.2% mRR/mAP overall respectively (with statistical significance). The other two JANUS combinations lead to mixed results: "Vis + Seq" leads to overall performance gains while "Txt + Seq" does not produce overall gains, due to its lower performance on the extended dataset.

When using "Visual + Textual", JANUS significantly outperforms TANGO on seven of nine apps and is only worse than TANGO on FCS, considering both mRR and mAP. As previously mentioned, JANUS's lower performance for FCS, compared to TANGO, stems from the nature of the app itself. FCS is a web browser and the bugs used for this app were not dependent on a particular web page. When reproducing the bugs, the users navigated to different web pages, each one having different layouts and appearances. This means that the duplicate video-based bug reports appeared to be substantially different. Since JANUS focuses more heavily on global GUI layout information, via its DINO+ViT model, JANUS struggles to differentiate duplicates from non-duplicates. The local features learned by TANGO seem to be useful for duplicate detection even when the duplicate videos show different layouts. The lower JANUS mAP value on GNU is explained by the lower mAP values of JANUS_{vis} and JANUS_{txt} on that app (by 0.4% and 1.2%—see Table 3.2).

JANUS's configuration "Vis + Txt + Seq" consistently shows mRR/mAP improvement in all nine apps except GNU, when compared to the baseline TANGO. Across these apps,

we observe improvements ranging from 6.8%/6.2% to 23.4%/26% mRR/MAP in the original dataset, and from 0.9%/1.8% to 7.5%/8.1% mRR/MAP in the extended dataset. This is interesting because the performance of the individual components of this configuration is substantially different across the apps. For instance, for TOK, the sequential aspect of the videos, individually combined with JANUS$_{vis}$ or JANUS$_{txt}$, is less effective than TANGO, but when JANUS$_{vis}$ and JANUS$_{txt}$ are combined together with JANUS$_{seq}$, JANUS leads to substantial improvement (by 6.8%/6.2% mRR/mAP). Another example is the FCS app, which seems to benefit from the visual and sequential information, as JANUS$_{vis}$ +JANUS$_{seq-v}$ seems to contribute most to the overall performance of the " Vis + Txt + Seq" configuration. This suggests that the incorporation of sequential information enhances the JANUS's ability to handle dynamic content, resulting in improved performance in comparison to its "Visual + Textual" configuration and the baseline TANGO.

Best Janus configuration: The best performing JANUS configuration is when combining visual (JANUS$_{vis}$), textual (JANUS$_{txt}$), and sequential information (JANUS$_{seq}$) from video-based bug reports. This configuration consistently outperforms the baseline duplicate detector for 8/9 mobile apps. It achieves an overall performance of 89.8%/84.7 mRR/mAP, outperforming the baseline by 8.7%/9.2% mRR/mAP. This means that JANUS can reduce the effort that developers spend determining if a new video-based bug report shows a known bug (by $(1.60 - 1.38)/1.38 = 16\%$, based on avg. rank), since they would need to inspect only 1.38 videos on average (*i.e.,* 1.38 avg. rank across all tasks) for finding the first duplicate video in the candidate duplicates suggested by JANUS.

3.5.5 Qualitative Analysis

We discuss two qualitative examples that illustrate the validity of our hypothesis that the richer representations learned by JANUS's transformer-based visual representation and OCR models improve duplicate detection for video-based bug reports.

3.5.5.1 Example 1: Vision Transformer-based Representations Capture Subtle GUI patterns

To illustrate why we observed improvement in visual JANUS as compared to visual TANGO, we use interpretability techniques that generate saliency maps that help visualize the learned visual features. To visualize patterns learned by CNNs, we use a technique called AGF [83]. Although AGF can visualize self-supervised models such as SimCLR (used by the baseline), this requires training a supervised linear classifier after each layer and a dedicated algorithm to extract the segmentation information from their weights. Therefore, to simplify our comparison, instead of visualizing SimCLR directly, we visualize its main component, the ResNet-50 CNN using AGF under supervision. We follow past work and use the pre-trained ResNet-50 (on ImageNet [59]: the training dataset for ResNet) to generate the saliency map based on the class IDs with the highest probabilities for a given target GUI screen [83]. We further visualized the ViT-S/16 model (used by JANUS$_{vis}$'S DINO) by directly displaying the self-attention maps. Visualization of ViTs does not necessitate sophisticated algorithms, given the inherent attention mechanism within these architectures.

In Fig. 3.2, we show three keyframes of two non-duplicate video-based bug reports from DroidWeight (DROID) [3]: DROID-CC1 and DROID-CC2. SimCLR fails to distinguish between the videos of these two bugs and mistakenly ranks DROID-CC2 as the first duplicate video of DROID-CC1. The DROID-CC1 video mainly has one trace that generates a new weight record by entering the weight on a pop-up component (Fig. 3.2 (b)), while the DROID-CC2 video not only includes the previous trace but also a trace that further edits the recorded weight on another different pop-up component (Fig. 3.2 (c)). Fig. 3.2 illustrates the saliency maps, overlaid over frames from two Droidweight video-based bug reports. We observe that the ViT-S/16 is able to attend to key parts of GUI components that ResNet-50 does not. Specifically, for the main screen (a) and *entering weight* screen (b) from videos of DROID-CC1 shown in Fig. 3.2, ResNet-50 and

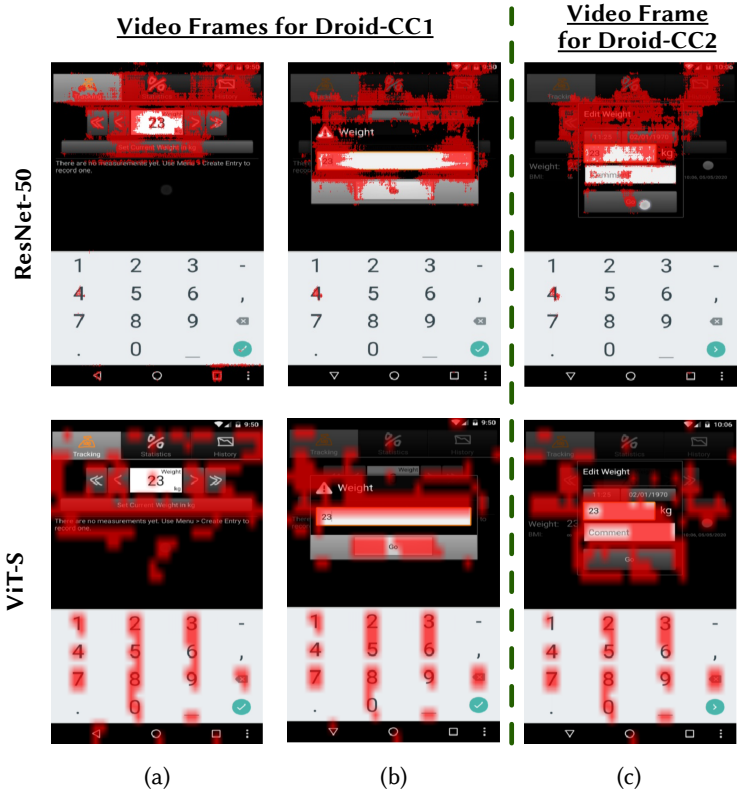

Figure 3.2: Visualization of ResNet-50 and ViT on keyframes of video-based bug reports

ViT-S/16 are all able to attend well to the objects, but ViT-S/16 pays more attention to the GUI layout information. However, for the *edit weight* screen (c) from videos of DROID-CC2, ResNet-50 has more difficulty in distinguishing between foreground pop-up components and the background. We can see it pays less attention to the lower edges and the bottom part of the foreground component. In contrast, ViT-S/16 effectively attends better to the edges and pays enough attention to the foreground component to help distinguish between (b) and (c), hence improving performance on this specific duplicate detection task.

From this example, there is a clear benefit to the visual nuances learned by ViTs. While here we present one example, after investigating several cases where JANUS$_{vis}$ outperforms the baseline, we observed this pattern holds, wherein JANUS$_{vis}$ learned visual representation is able to better capture nuanced visual patterns, such as the difference be-

tween two similar pop-ups, or the difference between background and foreground element when menus are displayed.

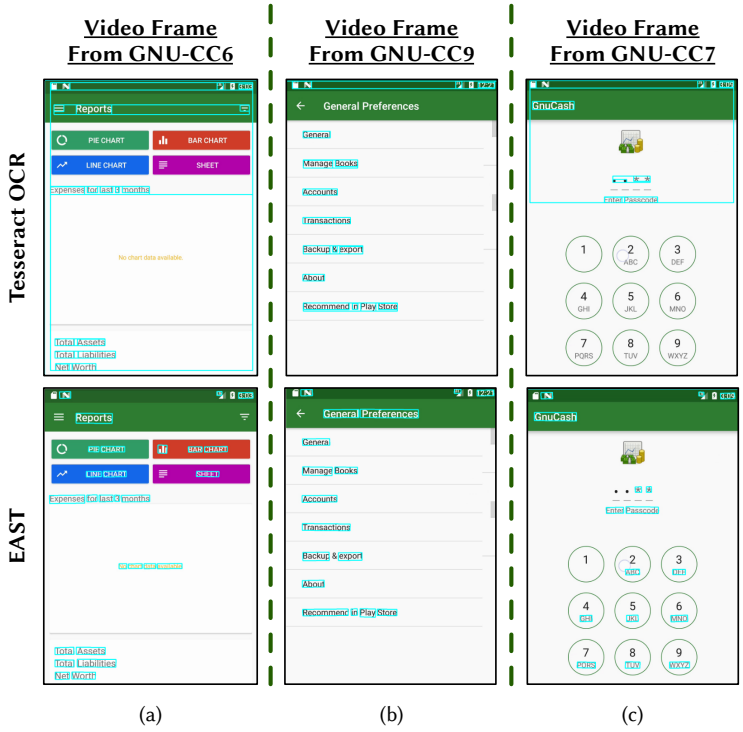

Figure 3.3: Bounding boxes localized by EAST and the Tesseract OCR library on keyframes of video-based bug reports

3.5.5.2 Example 2: Scene-based Text Detection Improves Text Localization

Textual TANGO, which uses Tesseract OCR is unable to distinguish between similar video reports for a number of bugs, including three bugs from the GNUCash (GNU) app [4]. Therefore, we visualize the detection bounding boxes of text for three keyframes of these three videos in Fig. 3.3 for both Tesseract (first row) and EAST [253] (used by JANUS). The first report for the GNU-CC6 bug has a main trace that goes to the `balance sheet` screen and checks the sub-account: we show one keyframe for this report in (Fig. 3.3-(a)), while the second video report for the GNU-CC9 bug navigates to the `General Preferences` screen, as shown in keyframe in (Fig. 3.3-(b)), and finally, the report for

GNU-CC7 changes the password under the `General Preferences` menu, as shown in (Fig. 3.3-(c)). While these bugs are different, they include many similar screens where keywords are important for differentiation.

As observed in Fig. 3.3, EAST is more accurate than TesseractOCR for GUI component and text detection. In Fig. 3.3-(a), Tesseract OCR fails to localize the text on some buttons (*e.g.,* sheet) and the text in brighter colors (*e.g.,* Asset). Also, for the keyframe of GNU-CC9 (Fig. 3.3-(b)), Tesseract misses the text `General Preferences`, making it difficult to distinguish between report GNU-CC9 and GNU-CC7, as they both access various parts of the settings menu. In addition, Tesseract does not detect the text when it is in regions of low brightness and low contrast, including the text on the dialing circles (Fig. 3.3(c)), which also helps differentiate between GNU-CC9 and GNU-CC7, since GNU-CC7 enters a passcode, but GNU-CC9 only accesses the passcode settings. Thus, the more accurate text extraction of EAST clearly aids in the accurate extraction of key text that can help to differentiate between similar GUI screens.

3.6 Threats to Validity

3.6.1 Internal and Construct Validity

Beyond the evaluation dataset, the implementation of Janus's models and experimental settings represent key validity threats. We controlled as many factors as possible for a fair comparison with the baseline. For example, we implemented the 4-Codebook approach in both Janus and the baseline, used the same duplicate detection tasks, and measured their performance using well-known metrics in duplicate detection studies.

3.6.2 External Validity

To improve generalization, we created a new dataset to include ≈3k more duplicate detection tasks, for real bugs of different types, reported on mobile app issue trackers. These bugs were recorded by multiple users on various versions of mobile operating systems and

did not include touch indicators. We ensured that the recorded videos contained different reproduction scenarios for the same bugs. Decisions were made to make our data set more comprehensive, realistic, and diverse. Our data set could be improved by considering different app languages or other mobile platforms such as iOS.

3.7 Chapter Summary

To help developers identify video-based bug reports that show identical bug reports in mobile apps, we propose Janus, a new approach to detect duplicate video-based bug reports. Janus leverages visual, textual, and sequential information from videos using a combination of representation learning, information retrieval, and frame alignment approaches.

We evaluated Janus and found that it significantly outperforms an existing duplicate detector. The evaluation considered a new benchmark of 7,290 duplicate detection tasks based on 270 video-based bug reports, drastically extending a prior dataset (with real bugs as opposed to injected bugs from prior work). We conducted ablation experiments and an in-depth qualitative analysis visually showing that Janus learns a more interpretable hierarchical visual representation and localizes text regions more accurately.

Chapter 4

Code Change Impact Analysis via Multimodal Coupling

Impact analysis (IA) is a critical software maintenance task that identifies the effects of a given set of code changes on a larger software project with the intention of avoiding potential adverse effects. IA is a cognitively challenging task that involves reasoning about the abstract relationships between various code constructs. Given its difficulty, researchers have worked to automate IA with approaches that primarily use coupling metrics as a measure of "connectedness" of different parts of a software project. Many of these coupling metrics rely on static, dynamic, or evolutionary information and are based on heuristics that tend to be brittle, require expensive execution analysis, or large histories of co-changes to accurately estimate impact sets.

In this project, we introduce a novel IA approach, called ATHENA, which combines the dependency graph information of a software system with a conceptual coupling approach that uses advances in deep representation learning for code without the need for change histories and execution information. Previous IA benchmarks are small, containing less than ten software projects, and suffer from tangled commits, making it difficult to measure accurate results. Therefore, we constructed a large-scale IA benchmark, from 25 open-

source software projects, that utilizes fine-grained commit information from bug fixes. On this new benchmark, our best performing approach configuration achieves a score of mRR, mAP, and HIT@10 of 60. 32%, 35. 19%, and 81. 48%, respectively. Through various calculations and qualitative analyzes, we show that ATHENA's novel combination of program dependence graphs and conceptual coupling information leads it to outperform a simpler baseline by 10.34%, 9.55%, and 11.68% with statistical significance.

4.1 Introduction

Modern software systems are long-lived and have extensive development and maintenance histories. Many projects experience churn in the developers or teams that work on them and can consist of millions of lines of code [192]. As such, understanding the potential cascading impacts of seemingly simple code changes can be a difficult proposition. This comprehension task forms the basis of *impact analysis (IA)* in which a given code change may result in *undesirable side effects*, such as a fault that leads to an erroneous program state, caused by unintended interactions between the changes and other parts of a software system [116,136]. Thus, the task of IA involves estimating an impact set of entities, usually classes or methods of a software system, from a given change to an entity, also usually a class or method [20] in the hopes of preventing unintended changes. This process can be cognitively challenging for developers, as reasoning about complex interactions of a software system requires careful comprehension of large volumes of code. Given that many important engineering and maintenance tasks – such as bug fixing and refactoring – require code change comprehension, they necessarily require IA as well. This process is typically performed *manually* by developers, but given its complexity, researchers have proposed a range of approaches to automate it.

Past techniques for automated IA have explored using four main types of information: (i) *structural information* (*i.e.,* from program dependence graphs), (ii) *semantic or conceptual information* (*i.e.,* code similarity), (iii) *evolutionary information* (*i.e.,* commit

histories), and (iv) *execution information*. Conventional automatic IA techniques [21, 30] have focused on analyzing structural dependencies (*e.g.,* control flow dependence) between different code entities to predict change impacts, but tend to generate large impact sets with lower precision [142]. As a result, other IA techniques have chosen to take advantage of additional information gathered by mining change histories from software repositories [39, 76] or program executions [136] to generate more accurate impact sets. However, these techniques rely on certain assumptions (*e.g.,* sufficient historical data, comprehensive execution profiles), require brittle heuristics, or significantly increase computational overhead, making them less practical. These techniques may also ignore conceptual / semantic information that naturally occurs in code (*e.g.,* identifiers) and is key to expressing the underlying intent of code entities. Given that code entities with similar intent likely contribute to similar problem domains, there is another set of IA techniques (*i.e.,* conceptual or semantic IA) [76, 116, 215] which extract vectorized code semantics and compute a similarity-based ranked list of code entities that are potentially impacted by a change. Existing conceptual techniques formulate IA as an information retrieval (IR) task, and typically apply IR-based (*e.g.,* latent semantic indexing (LSI)) or machine learning-based (*e.g.,* doc2vec [138]) approaches to obtain code representations that capture the semantic relationships between code entities.

The possibility of combining *semantic* and *structural* information specifically for the task of impact analysis has not been well explored [84]. Such a combination could prove beneficial due to the orthogonal nature of these information sources and the practicality of forgoing the collection and sanitation of evolutionary or execution information. For instance, semantic coupling can help to relate methods or classes that share similar semantic purposes and hence may impact one another, whereas structural information can help deduce logical relationships between code entities which may appear to be unrelated based upon modeled semantics.

While there is promise in combining semantic and structural information for IA, there is also an opportunity to leverage recent advances in robust semantic models of code.

Transformer-based [209] neural architectures [72,80,216,218] have achieved great success in learning rich representations for a variety of code understanding and generation tasks [162, 219], *e.g.,* code search, clone detection [223], program repair [222], *etc.*. These models are typically first pre-trained on large-scale datasets containing unimodal (code-only) and/or bimodal (comment, code) data to learn *generalized* code representations. The models are then fine-tuned on task-specific datasets for downstream code-related tasks. However, despite their demonstrated benefits, none of these models have been applied to IA.

However, adapting transformer-based models of code to the task of IA, and integrating these models with structural information presents at least two major challenges. First, we currently lack large-scale vetted data sets that would allow a neural model to be fine-tuned on *IA-specific* code representations. This is due to the fact that deriving an IA dataset is labor intensive, as impact sets cannot be easily extracted from software repositories without manual validation. Second, while the general code representations produced by pre-trained models could be directly used for similarity calculation for conceptual IA, they still ignore the global context the code finds itself in, *i.e.,* the structural dependencies that illustrate how the code is used within a software system. Unlike other code understanding tasks (*i.e.,* code search) that can rely solely on isolated code snippets to extract semantics, structural dependencies between code entities also play an important role in IA since mutually dependent entities are likely to be affected by each other.

To overcome these limitations and advance the task of automated IA, we introduce Athena, which enhances the understanding of the code with Transformer-based neural models [209] and structural dependence graphs to capture relationships between code entities. We perform IA at method-level granularity for code entities in the Java programming language (PL). Specifically, Athena begins by constructing a software system's dependence graph, where nodes represent methods, and edges represent the dependence relationship (*i.e.,* call dependence and class-member dependence) between methods. We then leverage neural code models including CodeBERT [72], UniXcoder [80], and GraphCodeBERT [81], prominent Transformer-based code models, for the initial extraction of

the embedding of the method. These pretrained neural code models are fine-tuned on a code understanding task, namely code search, to learn richer representations that are aware of underlying code intent and potentially transferring the additional knowledge learned from code search to IA. To integrate global dependence information into local code semantics, the initial method embeddings are further enhanced using an embedding propagation strategy inspired by graph convolutional networks (GCN) [127] based on the constructed dependence graphs.

Evaluating our proposed approach effectively also presents challenges. Existing IA benchmarks tend to be outdated and are constructed from original/unvetted commits, but as highlighted in multiple prior studies [129, 130, 164, 214], *tangling* has a high prevalence in these commits which is likely to affect the reliability of evaluation results of previous IA techniques on these benchmarks. Therefore, to evaluate ATHENA for the task of IA, we created a large-scale IA benchmark, called ALEXANDRIA, that leverages an existing dataset of fine-grained, manually untangled commit information from bug-fixes [97]. The benchmark consists of 910 commits across 25 open-source Java projects, which we use to construct 4,405 IA tasks – where each task consists of a query method and a set of impacted methods. Using the standard information retrieval metrics of mRR, mAP, and HIT@10, we find ATHENA significantly (based on statistical tests) improve over the best performed conceptual IA baseline by 10.34%, 9.55%, and 11.68% respectively.

4.2 Related Work

4.2.1 Impact Analysis Techniques

Typical IA techniques require a seed/starting entity to perform the analysis. Some start with a change request [76, 206] in natural language form, while most start with code entities [116,136,172] at different levels of granularity (*e.g.,* classes, methods, statements), since developers can usually identify at least one code entity that needs to be changed by using feature location techniques [61] and their knowledge about software development.

The output of the IA (*i.e.,* estimated impact set) is usually at the same granularity level as the seed entity. Given that the class / file level IA [206] is too coarse and the statement level IA [84] is too costly, most existing techniques choose to conduct IA at the method level [136,215]. Moreover, Java, as one of the most commonly used object-oriented programming languages (PLs), has been selected as the primary focus of IA more often than any other PL. (*e.g.,* C [84]).

In general, IA comprises two branches of techniques. One is to predict / infer the *potential* impact of all possible changes [35, 38, 84] (*i.e.,* dependence analysis); the other is to reason about the *actual* impact sets of code changes [116, 136, 215]. Specifically, the first branch assesses the user-perceived accuracy by creating the ground-truth impact set based on the static program dependence analysis or dynamic execution differencing, since they regard the real ground-truth as unknown. However, identifying the full set of dependencies based on static analysis is uncertain, and execution differencing relies on certain test cases and executions, which cannot cover all possible dependencies either. [33] gives a comprehensive summary of the first branch of techniques, while our approach falls into the second category, and we will now introduce the related techniques within this category in detail.

The existing IA techniques in the second category can be further divided into four types based on the information they analyze. *i.e.,* structural, conceptual/textual, evolutionary, or dynamic. Conventional IA approaches [21, 30] that use program graphs or slicing tend to generate very large impact sets [142], and most importantly, they ignore the conceptual information encoded in the code (*e.g.,* identifiers), which is also important for expressing the intent of code entities. Since code entities with similar intents likely contribute to similar problem/solution domains, conceptual IA techniques [116,172,206,215] typically apply IR-based (*e.g.,* LSI) or machine learning-based (*e.g.,* doc2vec [138]) approaches to code to extract vectorized code semantics and estimate impact sets by computing a ranked list based on cosine similarity of code entities. [172] quantitatively show that the conceptual coupling is superior to the structural coupling-based measures for IA. Moreover, some IA

techniques analyze evolutionary couplings [109, 190, 255] extracted from multiple historical releases/commits of version control systems to discover frequent co-change patterns to predict current change impacts, but sufficient historical data is not always available (*e.g.,* for new projects), and sometimes previous change patterns may be outdated and misleading. In addition, dynamic IA [29, 136] utilizes execution information (*e.g.,* execution traces, relations) to compute a more accurate impact set, but the computation overhead is much greater than static IA. The quality of dynamic techniques relies heavily on the representativeness of the test suites and/or profiles gathered during program execution. Industrial case studies [16, 28, 56, 84, 204] indicate preferences for static IA techniques over dynamic ones, as there is a lack of published studies reporting the adoption of dynamic IA [33].

To further improve the accuracy of the impact set estimation, some research attempts to combine existing techniques. [117] blend conceptual and evolutionary analysis showing additional advantages over using either of them alone. [76] further augment them with dynamic analysis to obtain more accurate impact sets. It should be noted that these two hybrid techniques are only compared with their variants (*i.e.,* using only one of the components) to validate the effectiveness. A recent work [136] combines dynamic analysis with structural analysis (*i.e.,* data and call dependencies) demonstrating that dynamic data sharing dependencies are complementary to dynamic call dependencies.

Our approach belongs to the set of hybrid analysis-based IA techniques, as ATHENA extracts the semantics of the code and dependencies and computes a ranked list to estimate the impact sets. Therefore, it avoids the associated limitations and drawbacks of other categories of techniques (*i.e.,* evolutionary and dynamic analysis) while retaining the benefits of multiple information sources. LSI is the model most commonly used to obtain code semantics for conceptual IA [76, 117, 172]. The latest and most closely related work to ours is [215] which integrates LSI with doc2vec to improve code semantics by considering the context of each code token within the code entity. They quantitatively show that the combined model outperforms using LSI only in IA.

Unlike existing conceptual IA techniques, our approach (i) leverages advanced transformer-based code models to obtain more meaningful code representations, and (ii) further enhances code semantics by embedding propagation based on structural dependence graphs. To our knowledge, our approach is the first IA technique that integrates global structural information into local code semantics based on only a single release of the source code without any additional information (*e.g.,* previous releases and/or execution information). Given that [215] has not made their implementation publicly available, we directly use LSI and doc2vec independently as conceptual IA baselines for our work. This also allows us to compare the performance of different models for code semantics extraction when they are individually applied for IA.

4.2.2 Impact Analysis Benchmarks

Existing IA benchmarks [35, 76, 136] are typically constructed in two ways. The first type of construction considers the impact sets of ground truth to be unknown and tries to create them using program dependence analysis [34, 36] or execution differencing [35, 37, 38, 84]. However, computing a full set of program dependencies [33] is an undecidable problem. As such, they are usually generated on the basis of artificial changes and/or by sampling changes in real open-source projects. All possible changes to a code entity (only involving *one* certain release of the code repository) are used as seeding entities.

The other more popular way for constructing IA benchmarks involves building multiple co-changed sets of code entities, each of which is collected based on *two* consecutive commits [136] or several grouped commits [215]. All entities within a cochanged set are assumed to be affected by each other. To construct the ground truth, one [116] or a few code entities [136] in the co-changed set are selected as the seed entity, and the remaining others are served as the real impact set. Existing benchmarks/case studies in this category usually consist of 3-6 open-source repositories, and the commits used are bug fixing commits only [111] or are dominated by bug fixing commits [76]. However, the prevalence of *tangling* [97,99,128,164] existing in commits negatively affects the reliability of the eval-

uation performance of techniques (*e.g.,* bug localization [164], defect detection [98]) that rely on commit data for testing due to the presence of noise. Tangled commits refer to the changes to software which address multiple concerns at once. For example, a (original) commit that claims to be fixing a bug may not only fix the bug, but also include additional unrelated changes (*e.g.,* refactorings). Although we have limited knowledge on the exact impact of tangled commits on the reliability IA technique evaluations, the potential for impact is clear — in tangled commits the co-changed code entities within a commit do not all contribute to a single concern (*e.g.,* bug fixing) and thus are not necessarily impacted by each other, leading to inaccurate ground-truth impact sets. Given that previous studies have confirmed the prevalence of tangled commits [97], it is highly likely that evaluations of past techniques were affected by this phenomenon.

Our ALEXANDRIA dataset falls into the second category of the IA benchmark, but with a *notable key difference* — it is built from untangled bug fixing commits [97]. Herbold *et al.*'s work quantitatively shows that tangled commits have a high prevalence, and the authors manually untangle them by annotating line-level change types. Using only co-changed code entities that have been manually verified to contribute to one concern (*i.e.,* bug fix), our benchmark contains more reliable ground-truth impact sets, and this favorable characteristic is demonstrated quantitatively through experiments. To our knowledge, our ALEXANDRIA is the first IA benchmark whose ground-truth impact sets are built from manually validated untangled commits. Moreover, ALEXANDRIA contains 910 commits from 25 systems, which is greater than the past benchmarks.

4.3 ATHENA

In line with previous conceptual IA techniques [76,116,215], we formulate impact analysis as an information retrieval task where if a developer intends to modify a method (*i.e.,* query/seed method) in a software system, ATHENA will return a ranked list of other methods that could be affected in descending order of likelihood. All methods, but the

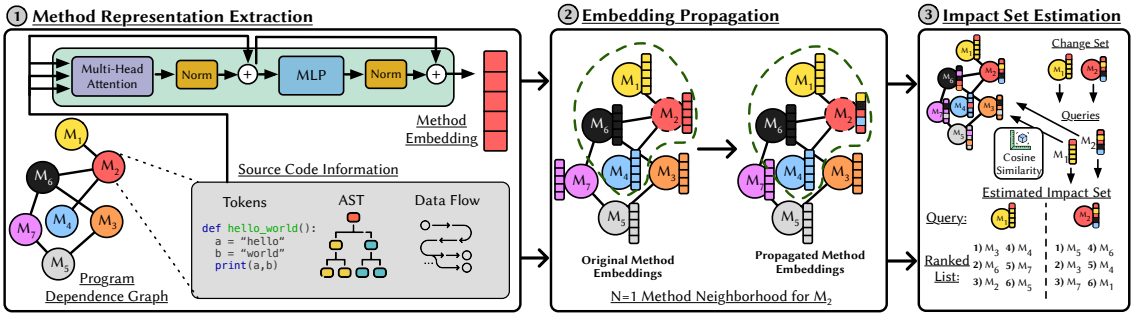

Figure 4.1: Overview of the Workflow of the Athena Impact Analysis Approach

query, are used as the search corpus. Formally, for a software system S containing a set of methods $S = \{m_1, m_2, ..., m_n\}$, a potential change to one of the methods $m_i \in S$ triggers ATHENA to rank all other methods, thus estimating the impact set.

Figure 4.1 provides an overview of ATHENA. ATHENA begins by building a dependence graph among all methods in a complete software system, where the nodes represent the methods and the edges represent the dependence relationships between the methods. Each method is processed by a state-of-the-art Transformer-based code model (*e.g.,* GraphCodeBERT) to obtain an initial method representation considering the context that exists within the method. These neural code models are then fine-tuned on the code search task to generate richer code representations and potentially transfer the additional knowledge learned from code search to IA. Next, ATHENA analyzes the global dependencies and propagates information from the nodes of the "neighbor" method in the dependence graph to a given target method. Specifically, each initial method embedding is updated/augmented based on a propagation strategy inspired by Graph Convolutional Networks (GCNs) [127] so that the information of global dependences is integrated into its local code semantics. To obtain a final ranking list, the cosine similarity between the augmented representations of a given query method and each method in the corpus is computed. Next, we discuss each step of ATHENA in detail.

4.3.1 Dependence Graph Generator

The initial step of ATHENA is to build a static dependence graph generator to capture method dependencies across a software system. Essentially, we identify two methods as having dependencies if there exists a caller-callee relationship between them (*i.e.,* call dependence) and/or if they belong to the same class (*i.e.,* class member dependence). Although certain existing tools such as WALA [73] and Soot [185] can produce static call graphs for Java, they require JVM bytecode as input, thus necessitating compliable source code. Although the latest version of Soot provides source code analysis, it limits the source code to Java 7 and still requires internal compilation. Thus, these tools increase the preprocessing time for IA and negatively affect their scalability. To better integrate the graph generator into ATHENA and capture the dependencies of both call and class members, we developed our own tool to generate static dependence graphs, which simply takes the source code of a software system as input.

A dependence graph can be formally defined as $G = (V, E)$, where V denotes a set of method nodes and E denotes a set of edges representing the method dependence relationships. Since impact analysis is usually performed on production entities (*i.e.,* excluding testing entities) [136], we first collect all `.java` production source files in a software system and use the Tree Sitter library [32] to identify all the methods contained in these files. The library enables the construction of a specific syntax tree for each file and supports the search for various patterns (*e.g.,* method calls, method declarations) in the tree. All identified methods then serve as the nodes of the dependence graph. To precisely locate each method and facilitate the process of method representation extraction, we attach to each method node the complete method content (*i.e.,* the declaration of the method with its body), the name of the class to which it belongs and the path of the package.

Next, we construct the edges for the dependence graph. To capture the class member dependencies, the edges are added between each pair of the methods in the same class. Regarding call dependencies, we utilize the Tree-Sitter library to identify all method in-

vocation statements (*e.g.,* `receiver.method()`) within each method and resolve these statements by finding its callee methods. The edges are then added between each pair of caller-callee methods. In general, we travel upward from each invocation statement to find where the `receiver` is introduced by analyzing the declaration statements and the arguments of the caller method. Then it is easy to obtain the class name of the callee method and its belonging package path. In order to locate the callee method based on the class name and the package path, we use both the method name and the # arguments (rather than the complete signature) to ensure the efficiency and scalability of our generator. When the callee method is overloaded with the same number of arguments, we add the edges from the caller method to each of these overloaded callee methods. It is worth noting that combining the method name and # arguments helps filter quite a few overloaded methods than using the method name only.

Although we can add directed edges from caller-to-callee methods, their semantics are actually interrelated and mutually affect each other when performing IA. Thus, by using our tool, the dependence graph is constructed in an undirected manner. Moreover, edges representing class member dependencies are distinguished from those representing call dependencies by attaching each edge to its property (*i.e.,* call or class member dependence). If two methods have both types of dependencies, we add two edges with different properties between them.

4.3.2 Code Representation Extraction

We then use one of three Transformer-based code models (CodeBERT, UniXcoder, or GraphCodeBERT) to extract the initial embeddings of the method to perform IA, as shown in Figure 4.1-①. In the case of GraphCodeBERT, it goes beyond the sequential information of the code by considering the inherent structure of the code (*i.e.,* data flow) to encode the relation 'where the value comes from' between variables. In this model, the input is encoded by a multilayer bidirectional transformer containing a sequence of self-attention and feedforward layers (*i.e.,* multilayer perceptron (MLP)) with normalizations.

These pretrained models can directly produce code embeddings, but the self-supervised objectives used during pre-training are quite different from IA, and most importantly, the representations are not specifically learned for Java, but generally for multiple PLs. Although these neural models can be further fine-tuned for downstream tasks, neither GraphCodeBERT nor other Transformer-based code models have been fine-tuned or evaluated for IA due to the absence of large available IA training/fine-tuning datasets. IA belongs to a general family of code understanding tasks (and hence is not generative), and there are two other downstream understanding tasks that have been extensively researched and evaluated, namely code search and clone detection. The code search aims to retrieve relevant code given an NL query, while clone detection aims to predict whether two code snippets can output similar results when given the same input. We leverage code search as a proxy to potentially transfer additional knowledge learned from code search during fine-tuning to enhance code semantics for IA. Although detection may initially seem more closely aligned with IA, we do not use it because (i) datasets such as BigCloneBench [158, 202] which could be used for fine-tuning do not include comments, which is likely to enhance code understanding; and (ii) instead of generating separate code embeddings, the fine-tuned neural model for clone detection concatenates two code snippets as a whole and only generate one embedding for them, thus making the following embedding propagation process more difficult. They typically add a classifier on top of the Transformer-based encoder to directly produce the probability of whether two code snippets can yield similar results.

To fine-tune our neural code models for code search, we follow the pipelines recommended in their corresponding papers. For example, for GraphCodeBERT, our best performing model, we follow the authors' recommendation [81] to use a Siamese framework on the CodeSearchNet [107] Java split dataset. CodeSearchNet consists of 2.3 million functions in six programming languages paired with NL descriptions (*i.e.,* comments). The CodeSearchNet Java split has been filtered by hand-crafted rules by [81] to remove low-quality data and contains 164,923 bimodal (comment, code) pairs. Each code snippet

in the paired data is a method from a software GitHub repository with all comments removed, and the corresponding comment is extracted from the first line of the method's documentation comment. The objective of fine-tuning is to map the code and its comment onto the vectors close to each other to learn high-level intent-aware code semantics. During fine-tuning, the comment and code (with data flow extracted) are separately fed into a comment encoder and a code encoder. These two encoders have identical model architectures (*i.e.,* GraphCodeBERT) and are initialized from the pre-trained GraphCodeBERT parameters (*i.e.,* weights and biases). The parameter updating is synchronized across both encoders during fine-tuning based on the standard cross-entropy loss. We use the AdamW [126] optimizer and the same hyperparameters (*e.g.,* # epochs, learning rate, batch size *etc.*) recommended by [81] for parameter updating, and the whole process was performed on an Ubuntu 20.04 server with an NVIDIA A100 40GB GPU. The fine-tuned GraphCodeBERT is expected to generate more meaningful representations of code that are aware of the underlying intent.

When performing IA, we need to first preprocess the method content attached to each method node in the generated dependence graph. Taking GraphCodeBERT as an example, we first follow the CodeSearchNet preprocessing procedure [107] by extracting the initial line of the documentation comment and the code-only data. The code is then parsed into an abstract syntax tree (AST), the leaves of which are used to identify the variable sequence for data flow construction. The input to the fine-tuned GraphCodeBERT for IA is the concatenation of the comment, the source code, the set of variables $X = ([CLS], A, [SEP], C, [SEP], V)$ or $X = ([CLS], C, [SEP], V)$. A, C and V stand for the comment token sequence, the code token sequence, and the variable sequence, respectively. $[CLS]$ is a token for learning aggregated information from the entire sequence during training, and its final representation is typically used for classification-related tasks. $[SEP]$ is a separation token that is used to split two types of data. Edges are added between variables in the variable sequence where a data flow relationship exists, and the variables are aligned across source code and data flow. The input is then processed by

the fine-tuned encoder, and we take the average output of all the hidden states of the last layer as the method representation. The input sequence length is set to 256 and the output representation dimension is 768 to maintain consistency with GraphCodeBERT. Finally, the initial method embeddings are generated for all the nodes of the method in the dependence graph of a given software system.

4.3.3 Embedding Propagation

Although the initial embeddings effectively capture meaningful code semantics via the self-attention mechanism, they are limited to the local context and lack the global dependence of methods. To further improve code understanding, we utilize an embedding propagation strategy that updates each embedding of the method by propagating the embeddings of its neighbor methods based on the constructed dependence graph G, thus integrating the information of global structural dependence into local code semantics. We visualize this process in Figure 4.1-②. Formally, this is represented as $m_i' = f(m_i, m_1^{nebr}, m_2^{nebr}, ..., m_k^{nebr})$, where m_i is the method that is being updated through the embedding propagation strategy f with its neighbors $m_j^{nebr}(1 \leq j \leq k)$. In particular, our embedding propagation strategy is inspired by the graph convolutional network [127] that adopts layer-wise propagation in neural networks motivated by a localized first-order approximation of spectral graph convolutions:

$$M' = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}MW), \tag{4.1}$$

where σ represents an activation function and W is a trainable weight matrix. $\tilde{A} = A + I_N$ denotes the adjacency matrix of a graph G with self-connections. I_N is the identity matrix and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. This propagation strategy has been modified using a renormalization method [127] to mitigate the effects of numerical instabilities and exploding / disappearing gradients when matrix multiplication operators are repeated during deep neural network training. Since we do not train our dependence graph G in this phase, our embedding

propagation strategy is directly derived from the first-order approximation of localized spectral filters on graphs [57, 87], which can be summarized as follows:

$$M' = (I_N + wD^{-\frac{1}{2}}(A^c + A^{cm})D^{-\frac{1}{2}})M. \tag{4.2}$$

$M \in \mathbb{R}^{N \times F}$ represents the matrix of all the method embeds with respect to the dependence graph G and $M' \in \mathbb{R}^{N \times F}$ stands for the matrix in which each method embedding is updated by its neighbor method embeddings. N denotes the number of nodes in the method and F denotes the dimension of each embedding method (*i.e.,* 768). A^c is the adjacency matrix based on the call dependence edges of G, while A^{cm} is the one based on class dependence edges. Neither of them contains self-connections. D is the degree matrix of $(A^c + A^{cm})$ for normalization with respect to both rows and columns. w is a constant that is responsible for balancing the information between methods and its neighbor methods. According to this formula, if a method exhibits both call and class member dependencies with its neighbor method, the embedding of this neighbor method will be propagated/aggregated twice to the target method embedding. Intuitively, methods that share multiple dependencies are inherently more closely related than those with only a single type of dependency. Moreover, in order to evaluate the effect of the distance of neighbor methods used for embedding propagation, neighbor methods in other orders(hops) are also utilized in addition to the direct neighbors:

$$M' = (I_N + w\sum_i D_i^{-\frac{1}{2}}(A_i^c + A_i^{cm})D_i^{-\frac{1}{2}})M, \tag{4.3}$$

where $1 \leq i \leq 3$ since we usually take into account neighbor methods within three orders due to computational constraints. After the embedding propagation strategy has completed, all the methods identified in a given software system will have an augmented embed calculated by propagating the *original* method embedding from neighbors to the target method, as illustrated at the top of Figure 4.1-③.

4.3.4 Impact Set Estimation

Finally, as illustrated in Figure 4.1-③, ATHENA calculates the cosine similarity between the augmented embedding of a given query method and the augmented embeddings of each of the methods in the search corpus. Based on the cosine similarity scores, ATHENA returns a ranked list in descending order to help developers find other methods that are possibly affected and likely to be modified.

4.4 Experimental Design

To evaluate the effectiveness of ATHENA in the impact analysis task, we investigate four research questions (RQ):

RQ_1: *How effective is* ATHENA *with / without embedding propagation compared to conceptual baselines on the impact analysis task?*

RQ_2: *How do call and class member dependencies improve* ATHENA'S *overall effectiveness in IA?*

RQ_3: *How well does* ATHENA *perform on IA based on different configurations (e.g., using other Transformer-based pre-trained code models)?*

RQ_4: *How does the tangled benchmark affect the reliability of IA evaluation results?*

RQ_5: *How do properties of different impact analysis tasks affect our studied techniques?*

4.4.1 Impact Analysis Benchmark: Alexandria

Our IA benchmark ALEXANDRIA is constructed from manually untangled bug fixing commits [97] in order to generate more reliable ground truth impact sets. Multiple prior studies [129, 168, 214], supported by manual validation, have consistently shown that tangled commits naturally occur in codebases. However, all existing IA benchmarks [116, 136, 215], built directly from these original/unvetted commits, inaccurately assume that all co-

changed entities in a commit address one single concern, thus impacted by each other. Invalidated data (*i.e.,* (query, ground truth impact set) pairs) are likely to be noisy, which can affect the reliability of experimental results of previous IA techniques.

Recently, Herbold *et al.* [97] introduced a large data set that covers 3,498 commits from 28 Java projects, with the purpose of studying the tangling that occurs in bug fixing commits. All selected projects are from the Apache Software Foundation and were developed by contributors from the open-source community or industry. These projects cover diverse application domains, such as build systems (*e.g., ant-ivy*), web applications (*e.g., jspwiki*), general purpose libraries (*e.g., commons*), *etc.*. In this dataset, each changed line was annotated with its type of change, whether it was modified to fix a bug, or it was a change to tests, whitespace, a documentation change, a refactoring or an improvement of unrelated features. The data were annotated by four participants, and consensus was obtained if at least three participants agreed on the annotation to ensure accuracy. Although some existing datasets [128, 130, 164] also manually untangle the commits, they either cover a limited sample of commits or typically perform untanglement at the commit or file level, which is relatively coarse-grained so that the validated co-changed entities cannot be identified at the method level. Therefore, we built our IA benchmark based on the fine-grained untangled dataset [97] allowing us to know exactly which methods are changed to address a single concern, thus generating a reliable ground truth for evaluation.

Co-Changed Set Construction. To create IA evaluation tasks, we systematically analyzed the data set from [97]. Using only co-changed code entities that have been rigorously manually verified to contribute to one concern, our benchmark ALEXANDRIA contains more reliable ground-truth impact sets. Specifically, for each line changed in production code files labeled as *'contributes to bug fix'*, we added the corresponding method to our benchmark by recording the information of GitHub Diff URL, repository name, commit ID, parent commit ID, file path, method name, line numbers indicating where the method starts and ends. Since [97] does not provide method-related information, such as method names and line numbers of method boundaries, we used the srcML library

[54] to locate each changed method based on the changed line numbers labeled. We utilized the snapshot/release of a software system that corresponds to the parent commit ID, as that is the state in which the change would be applied. Then, for each parent commit, we formulate a co-changed method set based on concurrently changed methods. Since there is no clear indication of a query/seed method, *i.e.,* which method would be changed "first" in the commit, we treat each method in the set of co-changed methods as a potential query, whereas the remaining others constitute the ground-truth impact set. From developers' point of view, they usually at least know where the change starts and intend to know which other methods need to be modified. We further post-process the dataset to exclude commits that contain only one changed method.

IA Task Definition and Settings Formally, for each set of co-changed methods $M = \{m_1, m_2, ..., m_n\}$, $n \geq 2$, we perform IA with a query being $\forall\ m_i \in M$ and the corresponding set of ground truth impact being $M - m_i$. We consider three different settings where the search corpus differs. In the first setting (**Setting 1 -whole**), the search corpus includes all methods except the query in all production files from the corresponding snapshot of the software system. This setting provides a comprehensive evaluation scenario in which all methods in the software system are taken into account. The similar process of formulating co-changed methods in IA tasks has been widely adopted in previous work to evaluate IA approaches [76, 116, 136]. In practice, conceptual IA techniques will generate a ranked list of methods in the corpus, and developers would determine whether a method should be modified by inspecting the corpus in the given order. After analyzing our benchmark, it was observed that methods in the same class are more likely to be changed together. To account for this and mitigate potential biases introduced by IA approaches that equally prioritize methods within the same class as the query, we formulate two more specific task settings. In our second setting, the methods in both the ground-truth impact set and the search corpus are of the same class as the query (**Setting 2 - inner**). In our third setting, the methods in both the ground-truth impact set and the search corpus are from different classes than the query (**Setting 3 - outer**).

Table 4.1: Dataset statistics of our evaluation benchmark

Settings	# queries	# commits	ground-truth set	corpus
1 - whole	4,405	910	15.14	3,346
2 - inner	3,379	734	4.47	30
3 - outer	2,999	444	17.21	3,440

Dataset Statistics. Two software projects (*i.e., santuario-java* and *wss4j*) in [97] are no longer accessible and for the software project *eagle*, we were unable to build any valid co-changed method sets, *i.e.,* the size of the co-changed set less than two. As a result, our benchmark contains 25 Java software projects, and the lines of code (LOC), # commits, # tasks for each project is shown in Table 4.5. Moreover, for each of the three settings, Table 4.1 shows # tasks, # commits, the average number of methods in the ground-truth impact set and in the search corpus respectively. Compared to Setting 2 (inner), which requires retrieving four or five affected methods out of 31 methods, Setting 3 (outer) is much more challenging, requiring 17 or 18 methods to be retrieved from a larger corpus with an average of 3,440 methods.

Tangled Counterpart. To analyze the effect of tangling commits on the evaluation of IA techniques, we also construct a benchmark without manually untangling similar to what previous IA benchmarks did [116, 136, 215]. Specifically, we directly construct co-changed method sets from original/tangling commits, so the bug fix changes are likely to be tangled with refactoring and unrelated improvement changes. Then, we compare the Alexandria dataset with its tangled counterpart in terms of tasks with inconsistent pairs (query, impact set). We observe that 606 tasks from 50 commits (setting 1 - Whole) in Alexandria could have brought inaccurate ground-truth impact sets if not untangling. Furthermore, the tangled Alexandria dataset has 856 tasks (out of 4,655) from 81 commits that are inaccurate with respect to (query, ground-truth impact set) pairs. The increase in the number of tasks and commits is due to an increase in the size of co-changed method sets, *i.e.,* more changed methods (for refactoring/unrelated improvement) are used as queries and some previously filtered commits with co-changed set less than two are likely to be

added again.

4.4.2 Evaluation Metrics

We use standard information retrieval metrics to measure the effectiveness of ATHENA, namely mRR (mean Reciprocal Rank), mAP (mean Average Precision) and HIT@k. For each task, the list of ranked lists generated by ATHENA is compared to the set of ground-truth impact data. Specifically, we calculated the *rank* of the *first* truly affected method found in the ranked list, indicating the number of methods developers need to inspect before finding the first that requires modification. The *reciprocal rank* is then calculated for each task and these values are averaged across all tasks to derive the final mRR score. Furthermore, we compute the AP score for each task and average these scores across all tasks to obtain the final mAP score. AP is the average of the precision values calculated after *each* method in the ground-truth impact set, which approximates the area under the uninterpolated Precision-Recall curve. mAP scores measure the ability of the approach to help developers identify all possible affected methods. Moreover, we use HIT@k to measure the proportion of successful tasks for the cut point k. A successful task means that the approach has found at least one truly affected method among the top-k results it returns.

Many IA techniques [136] rely on *Precision, Recall* and *F-measure* for evaluation since they consider IA as a binary classification task by finding possibly affected methods based on structural/evolutionary/dynamic dependencies. Therefore, what these techniques produce is not a ranked list, but an uneven impact estimate set, which is then directly compared to the ground truth impact set to compute an F score (*i.e.,* the harmonic mean of the *precision* and *recall values*). However, conceptual IA techniques [76, 116, 215], formulate IA as an information retrieval task but still adapted prior Recall/Precision/F-score metrics to the IR context. We argue that IR metrics provide a more realistic representation of the potential benefits that conceptual IA approaches may actually provide to a developer in a recommender system setting. Furthermore, the mAP score is more accurate

than the F measure because it analyzes *Precision-Recall* relationship globally rather than only based on the calculation of the mean value.

4.4.3 Baselines

We compare our approach ATHENA, with three baseline approaches that extract code semantics for intent-aware IA. Specifically, two traditional IR-based approaches (*i.e.,* TF-IDF and LSI) and a deep learning-based model (*i.e.,* doc2vec [138]) are used as our conceptual IA baseline. To use IR for IA, we first build a corpus using all production methods from a specific snapshot/commit of a software system. For each code token in a method, we calculate its term frequency (TF), which represents the number of times the token appears in the method, and the inverse document frequency (IDF), which is the number of occurrences of the code token in all code tokens from the corpus. Each method in the corpus is then represented as a TF-IDF vector for the following cosine similarity computation. In line with previous conceptual IA techniques [76, 215], LSI also employs singular value decomposition (SVD) on the TF-IDF matrix consisting of TF-IDF representations of all methods in the corpus, and the cosine similarity is computed based on the new dimension-reduced method representations. As for doc2vec, we first train the model using the *distributed memory* algorithm on the CodeSearchNet Java split dataset by concatenating comment tokens with code tokens to maintain consistency with the Transformer-based model (*e.g.,* GraphCodeBERT) training process. The doc2vec model can then generate representations based on the paragraph-based method for the constructed IA tasks.

4.4.4 ATHENA Configurations

By using our approach ATHENA, we integrate global dependence information into local code semantics to improve IA and set $w = 0.5$ for information balancing. We use GraphCodeBERT as the encoder for the final version of ATHENA, and in $\mathbf{RQ}_3$, as it achieves the best IA performance. We also validate the effectiveness of initial method represen-

tations (without embedding propagation) obtained by GraphCodeBERT for conceptual IA ($Athena_{ct}$) and conduct experiments by using call ($Athena_{ct+cd}$) or class member dependencies ($Athena_{ct+cmd}$) with GraphCodeBERT in order to quantitatively show the contribution of each type of dependence from the dependence graphs.

We also experimented with different encoders (*i.e.,* CodeBERT [72] and UniXcoder [80]) that are also fine-tuned on the code search task following the similar procedure described in section 5.3 in order to demonstrate the effectiveness of our approach when using other Transformer-based code models. Moreover, we try neighbors of different orders/distances (1-3) when propagating the embeddings based on structural dependence graphs. Additionally, we conduct experiments based on different initial method representations obtained by GraphCodeBERT, including with/without comment, using output of the [CLS] token to represent methods, and using the pre-trained GraphCodeBert directly without finetuning it on code search. Last, we also fine-tune the pretrained GraphCodeBERT on the BigCloneBench dataset [158] constructed for the clone detection task following the same procedure provided by [81], and employ this fine-tuned GraphCodeBERT to directly generate the probability of whether two methods are semantically similar for the IA task.

4.5 Evaluation Results

4.5.1 RQ$_3$: ATHENA Performance on IA

Table 4.2: Effectiveness of baseline models and ATHENA with different components

Baseline	Settings	mRR	mAP	Hit@10	Athena	Settings	mRR	mAP	Hit@10
TF-IDF	1-whole	49.57	25.38	70.35	ATHENA$_{ct}$	whole	52.38	28.86	73.87
	2-inner	73.86	64.69	94.61		inner	**75.94**	**66.24**	95.44
	3-outer	34.50	16.50	49.35		outer	40.39	21.43	58.19
LSI	whole	49.98	25.64	69.80	ATHENA	whole	**60.32**	**35.19**	**81.48**
	inner	74.11	64.97	94.53		inner	75.59	65.94	**95.80**
	outer	34.85	16.68	49.45		outer	**45.07**	**23.41**	**61.59**
doc2vec	whole	43.62	19.97	58.59	ATHENA$_{ct+cd}$	whole	54.26	30.43	76.96
	inner	68.93	59.05	90.97		inner	75.05	65.52	95.38
	outer	29.63	12.35	40.25		outer	42.50	22.70	60.95
LSI (+comment)	whole	50.28	26.16	70.94	ATHENA$_{ct+cmd}$	whole	59.55	34.50	80.50
	inner	73.83	64.69	94.61		inner	75.91	66.22	95.32
	outer	34.60	19.93	49.91		outer	42.93	22.02	59.92

Table 4.2 presents ATHENA's and baseline performance (%) on our ALEXANDRIA benchmark for IA. All of these models take code-only information (*i.e.,* without comment) as input except LSI (+comm.), and we will show the performance of Athena (+comm.) in the $\mathbf{RQ}_3$ Ablation Study. Table 4.2 reveal that the LSI model achieves the highest effectiveness in the baseline models across three settings. Given the effect of the number of related topics on LSI's performance, we experimented with varying numbers of related topics (for 0 to 2,000 in 100 increments) and selected the one with the best performance (1,300) for the final LSI configuration. Moreover, LSI only slightly outperforms TF-IDF on three metrics, indicating that the advantage is not significant if high-level code semantics is extracted through SVD. Surprisingly, the doc2vec model performs worse than LSI. This could be due to the fact that the IR-based approaches can directly build corpora and measure importance of code tokens on the evaluation dataset, and thus excel at keyword matching in favor of IA. However, for the deep learning-based model doc2vec, it is primarily trained for high-level semantics understanding rather than keyword matching with evaluation set unknown, but it struggles with understanding code intent compared to Transformer-based code models. In addition, we add comment information to the input for the best performing baseline LSI, but the with-comment version only performs slightly better than the one without comments in Setting 1 (whole), but not in Setting 2 (inner) and 3 (outer) on mRR and mAP, which does not result in the real improvement for IA. We provide detailed explanation of this in $\mathbf{RQ}_2$.

As can be seen from Table 4.2, both ATHENA$_{ct}$ (without embedding propagation) and ATHENA outperforms LSI with statistical significance (Wilcoxon's paired test $p < 0.05$) on three metrics across all settings, and their improvements in Setting 1 (whole) can mainly be attributed to the improvements in Setting 3 (outer). Specifically, ATHENA$_{ct}$ improves LSI by 2.4%/3.22% mRR/mAP in Setting 1, and 5.54%/4.75% mRR/mAP in Setting 3. In fact, LSI performs quite well in Setting 2 (inner) because of its proficiency in keyword matching and the observation that keyword overlap is more common among methods within the same class as the query. Yet the Transformer-based model GraphCodeBERT

excels in understanding the underlying code semantics, resulting in superior performance of ATHENA$_{ct}$ in both Setting 2 and 3. However, the improvements from Setting 2 and 3 do not all contribute to the performance gain for Setting 1. The reason behind this is that LSI tends to rank all methods in the same class as the query higher than those in other classes and methods in the same class are more likely to be actually affected as indicated by the ratio of ground-truth impact set size to the corpus size based on Table 4.1. Consequently, LSI results in better relative performance in Setting 1 (*i.e.,* smaller improvement margin got by ATHENA$_{ct}$) than in Setting 3, but this does not change the relative positions of methods within the same class (Setting 2) or methods in different classes (Setting 3). More evidence supporting this explanation is provided in **RQ**$_2$. In addition, when integrating global dependence information into local code semantics, ATHENA substantially outperforms LSI by 10.34%/9.55% and 10.22%/6.73% mRR/mAP in Setting 1 and 3 respectively. ATHENA considers neighbor methods within two orders (hops) in dependence graphs for embedding propagation.

4.5.2 RQ$_2$: The Impact of Call Dependence and Class Member Dependence

In Table 4.2, we also present the performance of ATHENA when utilizing either the call (*i.e.,*ATHENA$_{ct+cd}$) or the class member dependences (*i.e.,*ATHENA$_{ct+cmd}$) for embedding propagation based on dependence graphs, which allows us to investigate how each type of dependency contributes to the effectiveness of ATHENA in IA. By comparing both ATHENA$_{ct+cd}$ and ATHENA$_{ct+cmd}$ with ATHENA$_{ct}$, we observed that both of them outperform ATHENA$_{ct}$ and their improvements in Setting 1 (whole) are also attributed to the improvements in Setting 3 (outer). This confirms the accuracy of our dependence graph generator when capturing either the call or class member dependence.

Although ATHENA$_{ct+cd}$ and ATHENA$_{ct+cmd}$ obtain comparable results in Setting 2 and Setting 3, ATHENA$_{ct+cmd}$ outperforms ATHENA$_{ct+cd}$ in Setting 1 by 5.29%/4.07% on mRR/mAP. This is because in ATHENA$_{ct+cmd}$, the query method is integrated with

the information from all the other methods in the same class. As such it ranks all these methods higher than those in other classes, as previously described in Section 4.5.1. To further support this explanation, we experimented with another strategy for considering only class member dependence. Instead of using embedding propagation, we directly reduce the cosine distance of the query method and each method within the same class as the query by 50% for IA. The results are quite good in Setting 1 (60.72%/37.23% mRR/mAP), but as expected it behaves exactly the same as ATHENA$_{ct}$ in Setting 2 and 3 because while all methods in the same class are drawn closer to the query, the relative positions of methods in the same class or those in other classes remain unchanged. In addition, when comparing both ATHENA$_{ct+cd}$ and ATHENA$_{ct+cmd}$ with ATHENA, both contribute to ATHENA's effectiveness particularly in Setting 1+3.

4.5.3 RQ$_3$: Ablation Study

Table 4.3: Ablation Study of ATHENA on mRR and mAP

Settings	Encoders				# neighbor orders				[CLS] token		pretrain-only		+comm.		clone detect.	
	CodeBERT		UniXcoder		1 order		3 orders									
	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP
whole	58.40	33.37	60.19	34.61	59.42	34.33	59.90	34.73	56.36	32.10	59.92	32.86	59.92	34.96	47.26	22.72
inner	74.68	64.74	75.87	66.18	75.95	66.26	74.94	65.20	73.74	63.83	75.62	65.94	75.12	65.37	71.18	61.11
outer	43.09	22.08	43.93	22.64	43.80	22.56	44.66	23.12	42.67	22.12	41.48	19.99	45.11	23.54	32.42	14.43

Table 4.3 illustrates the various configurations of ATHENA for the ablation study. Specifically, we first conducted experiments using different pre-trained Transformer-based code models, namely CodeBERT and UniXcoder. Both of them were also fine-tuned on the code search task in order to transfer additional knowledge learned from code search to IA, similar to our approach with GraphCodeBERT. Also, we follow the procedures recommended in the corresponding papers for finetuning and IA evaluation (*e.g.,* AST is only used for UniXcoder pretraining, but not for finetuning and evaluation). Since CodeBERT only considers sequential code information during pretraining and finetuning, the method representations obtained by CodeBERT are not as meaningful as those obtained by GraphCodeBERT, which results in poorer performance than ATHENA on IA.

On the other hand, UniXcoder's IA results are comparable to GraphCodeBERT for IA in Setting 1 (whole), but it does not perform as well as GraphCodeBERT in Setting 3 (outer). This may be due to the fact that UniXcoder only utilizes AST information in pretraining, but not in finetuning and evaluation, unlike GraphCodeBERT, which utilizes data flow in all these phases, thus benefiting the understanding of the underlying code intent. Moreover, we experimented with neighbor methods of different orders (1 and 3) for embedding propagation for IA, and the results showed that utilizing neighbor methods within two orders (ATHENA) is the optimal choice. Although considering the third order involves more dependent methods and requires more computational resources, it does not improve the IA performance.

Moreover, instead of taking the average output of all hidden states from the final layer, we experimented with using the output of the [CLS] token of the Transformer-based model (*i.e.,* GraphCodeBERT) as the initial method representation for ATHENA. While the output of the [CLS] token is widely used for code understanding-related tasks (*e.g.,* code search), taking the average output of all hidden states is more suitable for representing code semantics for IA, according to the results showed in Table 4.2 and Table 4.3. We also conducted experiments by removing the code search fine-tuning of ATHENA and using the pre-trained GraphCodeBERT directly for initial method embedding extraction, but the pretrained GraphCodeBERT is less effective than the fine-tuned one (ATHENA) for IA especially in Setting 3 (by 3.59%/3.42% mRR/mAP). The reason is that during the code search finetuning, the code is mapped closer to its corresponding NL description, further enhancing the model's ability of understanding the underlying code intent and thereby improving ATHENA's effectiveness. In addition, we add the comment information to the input of ATHENA, but the benefit isn't obvious, probably because our IA evaluation benchmark ALEXANDRIA directly collect developer-written methods from commit history, resulting in some methods having (documentation) comments while others do not (in a realistic setting for IA), which may negatively affect the similarity computation between methods. However, the CodeSearchNet dataset used for code search fine-tuning is well-

curated to ensure each code snippet is paired with its corresponding NL description (*i.e.,* the first line of the documentation comment). Therefore, for the sake of efficiency, our final version of ATHENA takes code-only information as input with data flow extracted for IA.

Table 4.4: The results of LSI and ATHENA on the filtered ALEXANDRIA and its tangled counterpart.

Settings	LSI				ATHENA$_{ct}$				ATHENA			
	tangled		untangled		tangled		untangled		tangled		untangled	
	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP
whole	52.94	17.51	**58.42**	**18.88**	54.93	19.56	**60.55**	**20.86**	64.56	23.71	**68.36**	**24.88**
inner	80.72	70.36	**82.17**	**71.50**	81.37	69.03	**82.67**	**70.69**	81.81	69.16	**83.65**	**71.05**
outer	37.72	11.45	**42.89**	**12.80**	41.31	15.10	**46.06**	**15.72**	47.53	16.41	**50.65**	**17.09**

In addition, we replace code search with clone detection to use it as a proxy for IA. Specifically, we finetuned the GraphCodeBERT for clone detection following the same pipeline recommended by [81]. Instead of generating separate code embeddings, the model directly produces the probability of whether two code snippets can yield similar results, and as a result, the embedding propagation strategy cannot be applied. Therefore, we utilize the generated probability scores to obtain a ranked list for IA and compare it with ATHENA$_{ct}$ (without embedding propagation). However, from Table 4.2 and Table 4.3, we observe that using clone detection as a proxy is less effective than ATHENA$_{ct}$ using code search.

4.5.4 RQ$_4$: The Performance of ATHENA and the Baseline on the Tangled Benchmark Counterpart

In Table 4.4, we present the evaluation results of the best performing baseline LSI, ATHENA$_{ct}$, and ATHENA on the filtered ALEXANDRIA and its corresponding tangled counterpart using the mRR and mAP metrics. Specifically, after comparing our IA benchmark ALEXANDRIA with its tangled counterpart, we extract the tasks with inconsistent (query, ground-truth impact set) pairs and conduct experiments on these filtered tasks from ALEXANDRIA (untangled) and its tangled counterpart respectively. The statistics of the filtered datasets are described in Section 4.4.1. As observed in Table 4.4, there

is a significant performance difference between untangled ALEXANDRIA and its tangled counterpart across three settings when using any of the models, especially on mRR (ranging from 3.80% to 5.62% in Setting 1). However, existing IA benchmarks are typically built from tangled/original commits, which affects the reliability of evaluation results of previous IA techniques. Moreover, as expected, each of the three models perform better on untangled ALEXANDRIA than on the tangled version across 3 settings. The reason is that each co-changed set in ALEXANDRIA was manually verified to address one single concern, ensuring that methods within it are truly impacted by each other. In contrast, the tangled counterpart is built from original/unvetted commits and the methods within each co-changed set may not all contribute to one concern, thus not necessarily be impacted by each other. Therefore, Identifying the methods that are necessarily impacted with respect to the query is harder for each of the representative models.

4.5.5 RQ_4: Qualitative Analyses on Impact Analysis Tasks

Table 4.5: Effectiveness of ATHENA and the baseline (LSI) on each software system

Repo Name	LOC(k)	# Commits	# queries	Athena		LSI	
				mRR	mAP	mRR	mAP
ant-ivy	412.3	176	785	**50.19**	**26.47**	39.79	18.48
archiva	361.2	2	43	**70.81**	**32.17**	69.39	10.17
commons-bcel	168.3	18	138	**66.07**	**30.79**	57.76	21.68
commons-beanutils	67.5	11	42	65.64	**44.58**	**67.67**	43.64
commons-codec	55.1	8	41	**67.78**	**52.79**	57.65	34.91
commons-collections	136.3	15	73	**47.84**	**24.80**	41.43	18.85
commons-compress	147.3	61	260	**51.67**	**32.99**	45.26	23.73
commons-configuration	72.9	65	253	**56.89**	**36.87**	41.04	24.75
commons-dbcp	55.6	21	91	**67.17**	**52.55**	61.73	46.65
commons-digester	89.7	8	22	**38.65**	**29.07**	28.05	23.86
commons-io	102.5	19	58	**64.34**	**49.13**	52.16	32.53
commons-jcs	164	26	221	**70.35**	**26.10**	61.02	18.86
commons-lang	192.5	36	115	**67.16**	**56.23**	58.38	46.66
commons-math	431.1	124	589	**65.93**	**42.02**	52.43	29.20
commons-net	58.2	44	171	**66.59**	**44.59**	51.02	26.35
commons-scxml	43.8	28	114	**50.32**	**34.62**	45.82	31.19
commons-validator	42.3	12	35	**62.74**	**56.51**	51.70	40.29
commons-vfs	91.2	40	166	**55.02**	**36.62**	51.30	35.71
deltaspike	174.2	2	5	**60.98**	**57.65**	35.04	27.25
giraph	200.6	68	527	**70.80**	**38.40**	59.01	26.78
gora	132.4	40	174	**49.31**	**26.93**	41.91	23.59
jspwiki	439.4	1	12	87.50	40.03	**100.00**	**70.28**
opennlp	293.5	33	141	**64.61**	**40.16**	52.13	28.94
parquet	177.6	50	324	**60.09**	**25.92**	48.27	17.65
systemml	4000	2	5	**47.15**	**41.60**	41.63	31.25

We begin our analysis of IA tasks by looking at the performance of our studied

techniques across different studied software projects. Table 4.5 provides a finer-grained picture of the improvements per repository our ATHENA model achieves over the LSI baseline. As shown, ATHENA improves performance on 24 of 25 repositories in terms of mAP and 23 of 25 in terms of mRR in setting 1 (whole). For the failing repository *commons-beanutils*, we found that ATHENA substantially outperforms LSI in setting 3 (34.97%/30.58% vs.18.68%/12.37% mRR/mAP), but not in setting 2 (75.35%/64.92% vs. 88.37%/80.20% mRR/mAP). As for the repository *jspwiki*, it contains a single commit with 12 methods in the constructed co-changed set, which corresponds to 12 IA tasks. Among these 12 methods, 6 methods belong to one class, and the remainder are from another class. After investigating the failed tasks, we found that LSI was able to identify the affected methods quite well when the query and the ground truth methods had similar code lengths and a lot of keyword overlap, especially when they belonged to the same class. Now that we have examined the performance of ATHENA across IA tasks at a repository level, we will now discuss some exemplars from our benchmark that showcase how incorporating both structural information and semantic information can benefit IA.

Example 1: The Importance of Semantics. Figure 4.2 (a) shows two methods from different classes. The top method `checkStatusCode_URL_HttpURLConnection` from class `BasicURLHandler` is the query method and the bottom method `checkStat_URL_HttpMethodBase` is in the corresponding ground-truth impact set. This is representative of conceptual coupling [172], where the concepts of the two methods, *i.e.,* both performing a check on a status code, couples them together making it more likely that a change in one would result in a change in the other. Utilizing the semantic information between the methods, either through a traditional LSI or a Transformer-based neural model is necessary to determine that these two methods are highly related. Since they are not structurally dependent (via call or class member dependencies), structural dependence-only approach is likely to fail on this scenario.

Example 2: The Importance of Richer Semantics and Integration of Dependence Graphs. Figure 4.2 (a) illustrates a scenario with three methods from two

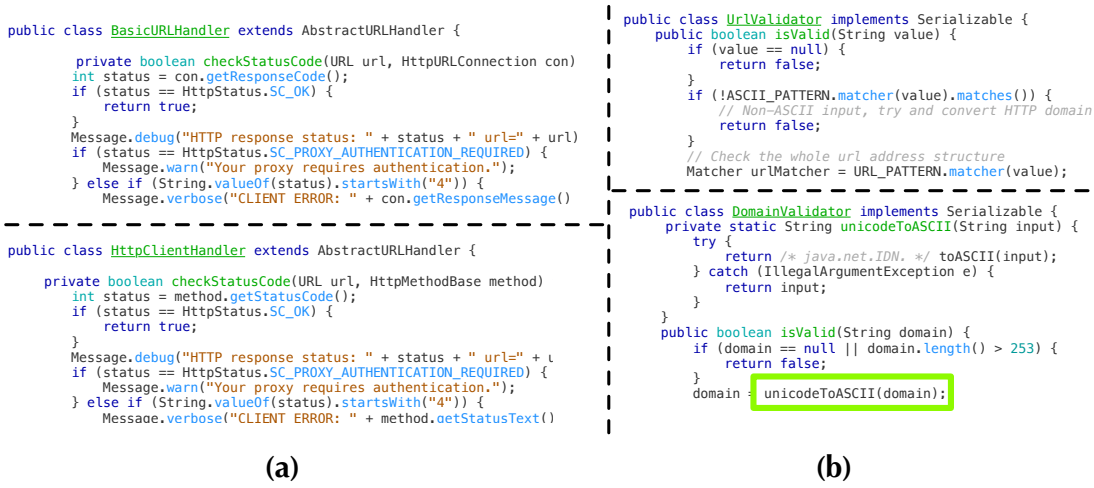

Figure 4.2: Two qualitative examples for illustrating the effectiveness of ATHENA.

different classes, where the method `isValid` from the class `UrlValidator` is the query, and the method `unicodeToASCII` and `isValid` from the class `DomainValidator` are in the ground-truth impact set. In this scenario, the baseline LSI ranks the `unicodeToASCII` method quite high at 589 due to the limited keyword overlap. When using ATHENA$_{ct}$ (without embedding propagation), which leverages GraphCodeBERT for better code understanding, the rank of the `unicodeToASCII` method improves to 137. However, it's still relatively high, which means developers might need substantial effort to locate this method. Remarkably, our ATHENA achieves a rank of 36, significantly outperforming the baseline. To understand why this occurred, we found that the method `isValid` in the `DomainValidator` class calls the `unicodeToASCII` method, which means these two methods have both call and class member dependencies. Through embedding propagation of ATHENA, the `unicodeToASCII` method is updated with information from the `isValid` method (in the `DomainValidator` class) that is more semantically similar to the query. This additional information helps improve the rank of the ground truth, even though there is no direct dependence relationship between the query and the `unicodeToASCII`.

As can be observed from these examples, there are clear benefits when code understanding is enhanced by the Transformer-based neural model and structural dependence graphs,

and we saw this pattern hold after investigating additional cases where ATHENA outperforms the baseline LSI. The contextual information obtained from the global call/class member dependencies among methods enriches the original semantics of the methods, which indeed helps to identify the impact set associated with the given query.

4.6 Threats to Validity

4.6.1 Internal Validity

To reduce potential issues from internal threats to validity, we experimented with three different DL models when validating our proposed approach of incorporating program dependence graph information into local code semantics to improve IA. Additionally, we constructed our benchmark from commits that have been manually annotated and had the changes made to fix bugs untangled from other changes, such as those to documentation, to ensure our benchmark is more reliable.

4.6.2 External Validity

To lessen the potential for threats to external validity, we used a significantly larger set of projects, 25 compared to previous work that used around five, and tested our approach across different DL models to show generalizability. One potential issue with generality is that we only evaluated our approach on Java and Apache projects; therefore, our approach may not generalize to other programming languages such as Python or to different types of project. However, the DL models we used have shown success across multiple programming languages, and so most likely the same would apply to our approach.

4.7 Chapter Summary

In this project, we introduce ATHENA, a novel technique for impact analysis that combines Transformer-based neural code semantics with structural dependence graphs. Addition-

ally, we established a large benchmark for impact analysis, which has been manually verified for bug-fixing commits. In our new benchmark, ATHENA demonstrates significant improvements over the simple conceptual baseline (+10.34% mRR, +9.55% mAP, and +11.68% HIT@10) and exhibits robust performance across software systems, with 23 of 25 systems showing improvement. Furthermore, our analysis reveals that the ATHENA's performance boost lies in its ability to more effectively identify impacted methods when they are outside the query method's class.

Chapter 5

Towards More Trustworthy Deep Code Models through Multimodal Out-of-Distribution Detection

Numerous ML models have been developed, including those for SE tasks, under the assumption that training and testing data come from the same distribution. However, training and testing distributions often differ as training datasets rarely cover the entire distribution, while the testing distribution tends to shift over time. Hence, when confronted with out-of-distribution (OOD) instances that differ from the training data, a reliable and trustworthy SE ML model must be capable of detecting them to either abstain from making predictions or potentially forward these OODs to appropriate models handling other categories or tasks.

In this paper, we develop two types of SE-specific OOD detection models, unsupervised and weakly supervised OOD detection for code. The unsupervised OOD detection approach is trained solely on in-distribution samples, while the weakly supervised approach utilizes a tiny number of OOD samples to further enhance the detection performance in various OOD scenarios. Extensive experimental results demonstrate that our proposed

methods significantly outperform the baselines in detecting OOD samples from four different scenarios simultaneously and also positively impact a main code understanding task.

5.1 Introduction

Extensive ML models have been developed under the assumption that training and testing data come from the same distribution (*i.e., closed-world assumption*). However, this assumption is often violated in practice, where deployed models often encounter out-of-distribution (OOD) instances that are not seen in training [207]. For example, a model trained on high-quality code may struggle to comprehend buggy code. Adapting ML models to distribution changes is possible, but challenging and costly due to constantly evolving data [152]. Moreover, even if the training data are up-to-date, models will still encounter unforeseen scenarios in the open world setting. Failure to recognize an OOD sample and, consequently, to produce incorrect predictions significantly compromises the reliability of a model. A reliable and reliable ML model should not only achieve high performance on samples from known distributions, *i.e.,* in-distribution (ID) data, but also accurately detect OOD samples, which can then either abstain from making predictions or potentially be forwarded to appropriate models handling other distributions or tasks.

OOD detection has been extensively studied in computer vision (CV) [235] and natural language processing (NLP) [145] on a range of tasks (*e.g.,* image / sentence classification, question answering). Existing OOD detectors typically design a scoring function to derive confidence/ID scores, allowing the detection of OOD samples based on a predefined threshold. These OOD detectors serve as an auxiliary function to the original ML models and ensure a high proportion (*e.g.,* 95%) [165] of ID data finally retained based on the threshold. This is crucial to prevent the OOD auxiliary scoring from adversely affecting ML models' performance on their main image/language-related tasks. Current OOD detection approaches are proposed in supervised, unsupervised, and weakly supervised

regimes depending on the availability of OOD data. The supervised approaches [95] learn a classical binary classifier based on both ID and OOD data, but in practice it is hard to assume the presence of a large data set that captures everything different from the ID data. Unsupervised ones [160, 252] only utilize ID data for training, but are likely to suffer from poor performance. Recent studies have shown that weak supervision [123, 124, 161, 205] can remarkably outperform unsupervised learning methods for the detection of anomalies / OOD. Some weakly supervised approaches [123, 124, 161] generate pseudolabeled OODs by partially corrupting ID data based on output attention mappings, while others [205] leverage a tiny collection of labeled OODs (*e.g.,* 1% of ID data) to detect specific OOD types in applications where access to OOD samples is limited and pseudo OOD generation is challenging [239]. However, none of these ML approaches have been applied in the context of SE for code-related tasks.

Existing OOD detection research in SE primarily focuses on anomaly detection or software defect detection. Anomaly detection techniques [82, 139, 149, 217] are designed to detect anomalous system states (*e.g.,* failed processes, availability issues, security incidents) during system running based on *monitoring data* (*e.g.,* logs, traces), but they still cannot been applied to the code context. There also exists a body of research dedicated to detecting suspicious defects in *source code* (*e.g.,* vulnerability detection [40, 189, 199], neural bug detection [18, 89]). Although defective source code represents a type of distribution shift from normal code, current defect detection techniques are not sufficient to cover a broad range of unseen scenarios considered by OOD detection.

Therefore, the goal of this project is to address the OOD detection problem in the context of SE for code-related tasks. Although transformer-based [209] NL-PL models have shown remarkable success in code understanding and generation [80, 81, 232] utilizing bimodal data (*i.e.,* comment and code), they often assume that training and testing examples belong to the same distribution. Thus, these models may not guarantee the robustness against OOD instances in the open world (as evidenced by [94] for NL Transformers). For instance, a code search engine, which is trained on GitHub comment-based

queries and code, is likely to fail in user questions and code answers from StackOverflow.

In this project, we systematically investigate the ability of pre-trained NL-PL models [80, 81, 150] in detecting OOD instances and the impact of OOD detection on a downstream code task (*i.e.,* code search). While NLP OOD detection techniques show promise for adaptation to NL-PL models due to the similarity between NL and PL, they can only detect textual OODs from uni-modal data. However, in the SE context for code-related tasks, distribution shifts can occur in either modality (comment or code) or both of them. An effective OOD code detector should be able to detect OOD from comments, code, or both modalities, by utilizing multi-modal NL-PL pairs. Several multi-modal approaches have been proposed for vision OOD detection [67, 165], utilizing information from both images and their textual descriptions, but they are still designed to detect *only* visual OODs.

To overcome these challenges, we develop two types of multimodal OOD detection models to equip NL-PL models with OOD code detection capability. The first is unsupervised (coined as COOD), which fine-tunes the NL-PL models to closely align NL-PL representations solely from ID data [107] based on multimodal contrastive learning [170], and then uses their prediction confidences as OOD scores. The contrastive learning objective is expected to effectively capture high-level alignment information within pairs (NL, PL) to detect OOD. To further enhance the OOD detection performance, we propose a weakly supervised OOD detection model, COOD+, which utilizes a tiny collection of OOD samples (*e.g.,* 1%) during model training. Current techniques in ML typically considered unsupervised contrastive learning [252] or outlier exposure [95,151], in conjunction with a scoring function, limiting their ability to detect OOD from only one modality. In contrast, our COOD+ integrates an improved contrastive learning module with a binary OOD rejection module to effectively detect OODs from NL, PL, or both modalities. OOD samples are then identified by a combination of two different scoring functions: the confidence scores produced by the contrastive learning module and the prediction probabilities of the binary OOD rejection module.

Due to the lack of evaluation benchmarks for OOD code detection, we create a new benchmark tailored for code context following the construction principles in ML [160,252], but containing more OOD scenarios: (1) aligned (NL, PL) pairs collected from a new domain, *e.g.,* from StackOverflow rather than GitHub, (2) misaligned (NL, PL) pairs, (3) the presence of syntactic errors in NL descriptions, and (4) buggy source code. We first evaluate the proposed models on two real-world datasets, CodeSearchNet-Java and CodeSearchNet-Python, and establish a range of unsupervised and weakly-supervised baselines for comparison. Experimental results show that both COOD and COOD+ models significantly outperform the best unsupervised and weakly-supervised baselines, respectively. Specifically, our unsupervised COOD is moderately capable of detecting OODs from three scenarios but does not perform well across all four scenarios. By integrating two modules, our COOD+ model effectively detects OODs from all scenarios simultaneously.

Furthermore, we apply our approaches to improve the robustness of existing (NL, PL) models for the code search task under the four OOD scenarios described above. By corrupting 15% of the testing dataset with OOD examples, we demonstrate that NL-PL models actually are not robust to OOD samples. Specifically, the performance of a fine-tuned GraphCodeBERT code search model drops by around 5% due to the presence of OODs. Subsequently, we filter the corrupted testing dataset with our COOD/COOD+, and show that our detectors successfully recover this performance loss and also improve the code search performance compared to the original testing set. In summary, the contributions are as follows.

- A novel OOD benchmark specifically designed for code contexts, encompassing multiple OOD scenarios;
- The first work to address OOD detection for code across four distinct scenarios;
- A multi-modal OOD detection framework for NL-PL pre-trained models, leveraging contrastive learning in both unsupervised and weakly-supervised settings;

- A comprehensive evaluation showcasing the superior performance of our COOD and COOD+ frameworks in detecting OOD samples across four scenarios;
- An online appendix providing the full codebase and experimental infrastructure of our approaches [233].

5.2 Related Work

We review related work on OOD detection in various fields such as CV, NLP, and SE, and then point out the unique characteristics of our approach.

5.2.1 OOD Detection in SE

To ensure the reliability and safety of large-scale software systems, extensive work [139, 240, 245] has been carried out on anomaly detection to identify anomalous system state (*e.g.,* failed processes, availability problems, security incidents) during system operation based on `monitoring data` (not in code format). Specifically, monitoring data include logs [139, 217], metrics (*e.g.,* response time, CPU usage) [243], traces [82, 149], etc. Although some approaches utilize supervised learning techniques [159, 247], others employ unsupervised [68] or semisupervised learning [140, 236] due to insufficient anomaly labels. However, none of these anomaly detection techniques targets code-based OOD detection, the main focus of our work. We mention this research line here since some existing OOD-related work in ML uses the terms *anomaly detection* and *(generalized) OOD detection* interchangeably [95], but anomaly detection in SE has distinct characteristics as described.

Furthermore, current defect detection techniques [158] in SE typically identify defects by analyzing `source code` with extracted code semantic features. Research in vulnerability detection focuses on security-related defects, such as buffer overflows and use-after-free. Compared to conventional static tools [7, 8], DL-based techniques [40, 154, 189, 199] utilize graph neural networks (GNN) or transformers to learn implicit vulnerability patterns from the source code. Additionally, bug detection techniques [18, 51, 89, 119] also fall under the

umbrella of defect detection but typically address semantically incorrect code (*e.g.,* wrong binary operators, variable misuse) which is not necessarily security-related and probably syntactically feasible. Although our focus is also on the source code, defective code is only considered as one scenario within the scope of our OOD detection problem.

More recently, several studies have explored the robustness and generalization of source code models to different OOD scenarios [86, 103, 221]. Hu *et al.* introduced a benchmark dataset to assess the performance of code models under distribution shifts [103], while others investigated fine-tuning strategies like low-rank adaptation [86] and continual learning [221] for enhanced *generalization* on OOD data. However, these studies did not specifically tackle OOD *detection*, and existing unsupervised OOD detectors have shown limited effectiveness for source code data [103]. In short, unlike previous work, our study directly aims to improve the OOD detection performance of existing code-related models, ensuring greater robustness and trustworthiness in the open world where many unseen OOD scenarios may be encountered.

5.2.2 OOD Detection in CV and NLP

In the ML community, the detection of OOD [93,145,186,235] has been extensively studied over the years, leading to a better defined and formulated task. The primary objective of OOD detection here is to design an auxiliary ID-OOD classifier derived from neural-based visual and/or textual models based on OOD scores. Given that correctly predicted instances tend to have higher maximum softmax probabilities (MSP) than incorrectly predicted and OOD instances, the MSP-based OOD scoring function [91, 93] was initially used to identify OOD samples. Subsequently, energy and distance-based scores [151, 201, 252] have also been used to derive OOD scores. For visual OOD data, existing techniques often aim for multiclass classification tasks (*e.g.,* image classification) and learn a $K + 1$ classifier assuming that the unseen space is included in the additional class [104,156]. The OOD data utilized for evaluation are typically constructed from a completely different dataset (out-domain data) or by holding out a subset of classes in a categorized dataset,

where one category is considered normal and the remaining categories are treated as OOD.

In the context of textual data, OOD detection techniques are applied to both classification tasks (*e.g.,* sentiment/topic classification [123, 252]) and selective prediction tasks [118, 208, 228] (*e.g.,* question answering, semantic equivalence judgments). These techniques rely on various algorithmic solutions including exposure to outliers [104, 242], data augmentation [244, 251], contrastive learning [112, 252], *etc.*. Compared to traditional neural-based language models, pre-trained Transformer-based [209] models exhibit greater robustness to distributional shifts and are more effective in identifying OOD instances [94, 229]. In addition to the out-domain data, text-based OOD detection also considers syntactic OOD data [160] due to the intrinsic characteristics of sentences. Syntactic OOD and ID data come from the same domain, but the syntactic OOD data have its word order shuffled, which allows for the measurement of OOD detectors' sensitivity to underlying syntactic information while preserving word frequency.

Some studies [200,213] have explored the incorporation of multimodal data into neural-based models to improve OOD detection accuracy. Recently, CLIP-based methods [67,74, 165] have emerged as a promising approach to OOD detection utilizing vision-language bimodal data, showing superior performance over unimodal data only. The main intuition behind these approaches is to take advantage of the alignment between visual classes or concepts and their textual descriptions. For example, Ming et al. [165] detect visual OOD in an unsupervised manner by matching visual features with known ID concepts in the corresponding textual descriptions.

However, these studies typically focus on detecting OOD data from at most two scenarios (*i.e.,* out-domain and shuffled text OODs) within a *single* modality. Even multimodal approaches are often limited to detecting only visual OODs by additionally considering accompanying textual descriptions. Our proposed approach aims to effectively identify OOD samples from four different scenarios in *two* modalities (*i.e.,* NL and PL). To achieve this, we use a combination of different scoring functions from two different modules: cosine similarities of a contrastive learning module and prediction probabilities of a

binary OOD classifier.

5.3 Approach

In this section, we first formally define the OOD code detection problem for (NL, PL) models (Sec. IV-A), then introduce the overall proposed framework (Sec. IV-B), and finally present details of unsupervised COOD and weakly-supervised COOD+ in Sec. IV-C and Sec. IV-D, respectively.

5.3.1 Problem Statement

Since current state-of-the-art code related models [80,81] typically extract code semantics by capturing the semantic connection between the NL modalities (*i.e.,* comment) and PL (*i.e.,* code) modalities, we formally defined OOD samples involving these two modalities in the SE context following the convention in ML [235,252]. Consider a dataset comprising training samples $((t_1, c_1), y_1), ((t_2, c_2), y_2), ...$ from the joint distribution $P((T, C), Y)$ over the space $(\mathcal{T}, \mathcal{C}) \times \mathcal{Y}$, and a neural-based code model is trained to learn this distribution. Here, $((t_1, c_1), y_1)$ represents the first input pair of (comment, code) along with its prediction of ground truth in the training corpus. T, C and Y are random variables on an input (comment, code) space $(\mathcal{T}, \mathcal{C})$ and a output (semantic) space $\mathcal{Y}$, respectively. OOD code samples refer to instances that typically deviate from the overall training distribution due to distribution shifts. The concept of distribution shift is very *broad* [186, 235] and can occur in either the marginal distribution $P(T, C)$, or both $P(Y)$ and $P(T, C)$.

We then formally define the OOD code detection task following [102, 123, 151, 205] as follows. Given a main code-related task (*e.g.,* clone detection, code search, *etc.*), the objective here is to develop an *auxiliary* scoring function $g : (\mathcal{T}, \mathcal{C}) \rightarrow \mathcal{R}$ that assigns higher scores to normal instances where $((t, c), y) \in P((T, C), Y)$, and lower scores to OOD instances where $((t, c), y) \notin P((T, C), Y)$. Based on whether to use OOD instances during the main-task training of pre-trained NL-PL models, we define OOD for code in

two settings, namely unsupervised and weakly supervised learning. For the unsupervised setting, only normal data are used in the main-task training. Conversely, weakly supervised approaches utilize ID and a tiny collection of OOD data (*e.g.*, 1% of ID data) [205] in training. In this context, the output space $\mathcal{Y}$ is typically a binary set, indicating normal or abnormal, which is probably unknown during inference. Due to the small number of training OOD data, the OOD samples required by our COOD+ and other existing weakly supervised approaches [184, 205] in ML can be generated at minimal cost and feasibly verified by human experts when necessary.

5.3.2 Overview

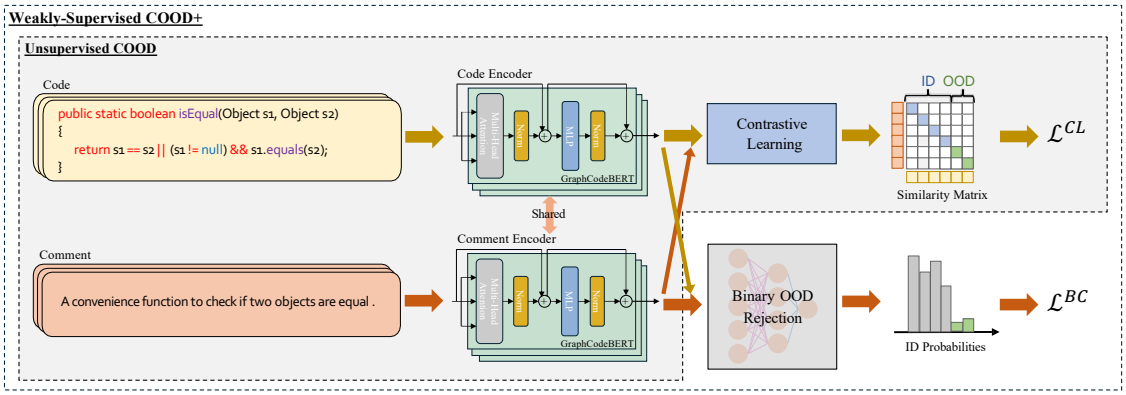

Figure 5.1: The Overview of Our Proposed COOD and COOD+ Approaches for OOD Detection

Overall, there are two versions of our COOD approach: unsupervised COOD and weakly supervised COOD +. Given a multimodal input (NL, PL), the unsupervised COOD learns distinct representations based on a contrastive learning module utilizing a pre-trained Transformer-based code representation model (*i.e.*, GraphCodeBERT [81]). Then, these representations are mapped to distance-based OOD detection scores in order to indicate whether the test samples are OODs during inference. The weakly supervised COOD+ further integrates an improved contrastive learning module with a binary OOD

rejection module to enhance the detection performance by using a very tiny number of OOD data during model training. The OOD samples are then identified by the detection scores produced by the contrastive learning module, as well as the prediction probabilities of the binary OOD rejection module.

5.3.3 Unsupervised COOD

Our unsupervised COOD approach consists of a contrastive learning (CL) module trained only on ID samples. Specifically, given (comment, code) pairs as input, we fine-tune a comment encoder and a code encoder through a contrastive objective to learn discriminative features, which are expected to help identify OOD samples based on a scoring function.

The (comment, code) pairs are first converted into the comment and code representations, which are processed by the comment and code encoder, respectively. We use the pre-trained GraphCodeBERT model [81] as the encoder architecture (*i.e.,* backbone). GraphCodeBERT is a Transformer-based model pre-trained on six PLs by taking the (comment, code) pairs as well as the data flow graph of the code as input, which has shown superior performance on code understanding and generation tasks. All the representations of the last hidden states of the GraphCodeBERT encoder are averaged to obtain the sequence-level features of comment and code.

Contrastive Learning Module. To achieve the contrastive learning objective, we fine-tune the base (GraphCodeBERT) encoders with the InfoNCE loss [170]. The comment and code encoders follow the Siamese architecture [80] since they are designed to be identical subnetworks with the same GraphCodeBERT backbones, in which their parameters (*i.e.,* weights and biases) are shared during fine-tuning. Parameter sharing can reduce the model size and has shown state-of-the-art performance for the code search task [191]. To extract discriminative features for (comment, code) pairs, we organize them into functionally-similar positive pairs and dissimilar negative (unpaired) pairs. Through a contrastive objective, positive pairs are drawn together, while unpaired comment and

code are pulled apart. Specifically, for each positive (comment, code) pair (t_i, c_i) in the batch, the code in each of other pairs and t_i are constructed as in-batch negatives, similarly for the comment side. The loss function then formulates the contrastive learning as a classification task, which maximizes the probability of selecting positives along the diagonal of the similarity matrix (as shown in Fig. 5.1) by taking the *softmax* of projected embedding similarities across the batch. The loss function can be summarized as follows:

$$\mathcal{L}^{CL} = -\frac{1}{2N}(\sum_{n=1}^{N} \log \frac{e^{sim(v_{t_i}, v_{c_i})/\tau}}{\sum_{j=1}^{N} e^{sim(v_{t_i}, v_{c_j})/\tau}} + \sum_{n=1}^{N} \log \frac{e^{sim(v_{t_i}, v_{c_i})/\tau}}{\sum_{j=1}^{N} e^{sim(v_{t_j}, v_{c_i})/\tau}}) \tag{5.1}$$

where v_{t_i} and v_{c_i} represent the extracted features of the comment t_i and the code c_i. τ is the temperature hyperparameter, which is set to 0.07 following previous work [191]. $sim(v_{c_i}, v_{t_i})$ and $sim(v_{t_i}, v_{c_j})/sim(v_{t_j}, v_{c_i})$ represent the cosine similarities between comment and code features for positive and negative pairs, respectively. N is the number of input pairs in the batch. InfoNCE loss is designed for self-supervised learning and learns to distinguish positive pairs from in-batch negatives. Compared to other contrastive losses [122, 252], it can take advantage of large batch size to automatically construct many diverse in-batch negatives for robustness representation learning, which is more effective to capture the alignment information between comment and code.

Scoring Function. Existing OOD detection techniques in ML derive scoring functions based on model's output, which typically map the learned class-probabilistic distributions to OOD detection scores for testing samples. Maximum Softmax Probability (MSP) [92] is commonly used for OOD scoring. This method uses the maximum classification probability $\max_{l \in L} \textit{softmax}\ (f(v_t, v_c))$, where $f(v_t, v_c)$ is the output of the classification model, with low scores indicating low likelihoods of being OOD. However, NL-PL code search models typically utilize similarity retrieval scores of NL-PL output representations to make predictions. Therefore, to enable simultaneous similarity and OOD inference, we alternatively extract cosine similarity scores for the testing of NL-PL pairs as OOD detection scores, denoted as $P^{CL} = sim(v_c, v_t)$. The underlying intuition behind

this scoring metric is that OOD testing samples should receive low retrieval confidence from the model fine-tuned on ID data, which establishes a closer relationship between ID (comment, code) pairs. Hence, this scoring function also assigns higher scores to ID data and lower scores to OOD data, similar to previous scoring methods.

5.3.4 Weakly-Supervised COOD+

To further enhance the performance of unsupervised COOD, we extend it to a weakly supervised detection model, called COOD+, which takes advantage of a few OOD examples. Inspired by [65], our COOD + combines improved contrastive learning (CL) and a binary OOD rejection classifier (BC). The improved CL module adopts a margin-based loss [230] which enforces a margin of difference between the cosine similarities of the aligned and unaligned pairs (comment, code) and restricts the cosine similarities of OOD pairs below another margin. The BC module integrates features from both comments and code to calculate the probabilities of OOD pairs. The OOD scoring function is then designed by combining the cosine similarity scores from the CL module and the prediction probabilities from the BC module. Below, we detail each component of our weakly supervised COOD+.

Improved Contrastive Learning (CL) Module. Given a batch of N input pairs (comprising $N-K$ ID pairs and K OOD pairs), latent representations are first obtained from the comment and code encoders. The margin-based loss is then leveraged in the CL module to distinguish representations of ID and OOD data by constraining the cosine similarity. Specifically, the margin-based contrastive loss is first applied to the $N-K$ ID code to maximize the difference between aligned pairs (comment, code) and incorrect pairs for each batch:

$$\mathcal{L}^{ID} = \sum_{i=1}^{N-K} \left(\frac{1}{N} \sum_{j=1, j\neq i}^{N} \max\left(0, m - s(v_{t_i^+}, v_{c_i^+}) + s(v_{t_j^-}, v_{c_i^+})\right) \right) \tag{5.2}$$

$s(v_{t_i^+}, v_{c_i^+})$ represents the cosine similarity of representations between each aligned ID

pair from all the $N-K$ aligned pairs, and $s(v_{t_j^-}, v_{c_i^+})$ represents the cosine similarity of representations between each ID code and all the other $N-1$ comments (*i.e.,* the comment is either not aligned with the ID code or from OOD comments). Thus, this margin-based loss encourages the difference between the aligned pairs and the incorrect pairs greater than margin m.

Regarding the K OOD code, we enforce a constraint on the cosine similarity between each OOD code and all the comments, ensuring that the similarity remains below a margin m. This constraint is necessary because each OOD code should not align with its corresponding comment, nor with any of the other $K-1$ OOD comments and the $N-K$ ID comments. The loss function is denoted as follows:

$$\mathcal{L}^{OOD} = \sum_{k=1}^{K} \left(\frac{1}{N} \sum_{i=1}^{N} \max \left(0, -m + sim(t_j^-, c_k^-) \right) \right), \tag{5.3}$$

where $sim(t_j^-, c_k^-)$ represents the cosine similarity between each of the K OOD code and all N comments. Finally, the overall loss for the contrastive module can be expressed as:

$$\mathcal{L}^{CL} = \frac{1}{N} \left(\mathcal{L}^{ID} + \mathcal{L}^{OOD} \right). \tag{5.4}$$

Binary OOD Rejection (BC) Module. Besides the CL module, we also introduce a classification module under weakly-supervision for identifying OOD samples. Inspired by the Replaced Token Detection (RTD) objective utilized in [72], we bypass the generation phase since our OOD data are generated prior to training. Therefore, we directly train a rejection network responsible for determining whether (comment, code) pairs are OOD or not, which can be framed as a binary classification problem. Our binary OOD rejection network comprises a 3-layer fully-connected neural network with *Tanh* activation, and the input is based on the concatenation of features from the comment and code encoders: $v_i = (v_{t_i}, v_{c_i}, v_{t_i} - v_{c_i}, v_{t_i} + v_{c_i})$. Apart from utilizing the comment and code features, we also incorporate feature subtraction $v_{t_i} - v_{c_i}$ and aggregation $v_{t_i} + v_{c_i}$. Additionally, we apply the sigmoid function to the output layer, producing a prediction probability that

indicates whether the sample is OOD. We then use binary cross entropy loss for this module:

$$\mathcal{L}^{BC} = \frac{1}{N} \sum_{i=1}^{N-K} (y_i \log p(v_i) + (1 - y_i) \log(1 - p(v_i))), \tag{5.5}$$

where $p(v_i)$ is the output probability of the BC module, and $y_i \in [0, 1]$ is the ground-truth label. $y_i = 1$ indicates the input sample is an inlier, while $y_i = 0$ signifies it is an outlier.

Hence, for weakly-supervised COOD+, we combine the objectives of the CL and the BC modules to jointly train our model, where λ is a weight used to balance the loss functions:

$$\mathcal{L} = \mathcal{L}^{CL} + \lambda \mathcal{L}^{BC}. \tag{5.6}$$

Combined Scoring Function. Similar to the unsupervised COOD approach, we utilize the diagonals of the similarity matrix as the OOD detection scores obtained from the CL module. To further improve the detection performance of the weakly-supervised version, we combine these P^{CL} scores with the output probabilities of the BC module, denoted as P^{BC}. Here, we convert cosine similarity scores into probabilities using the sigmoid function $P^{CL*} = \sigma(sim(v_c, v_t))$, then use multiplication to create the overall scoring function, yielding $P^{ID} = P^{CL*} \times P^{BC}$. We anticipate that higher scores will be assigned to ID pairs, while lower scores will be assigned to OOD pairs. This combined scoring function aims to enhance the discrimination between inliers and outliers, leading to more effective OOD detection.

5.4 Empirical Evaluation Design

To evaluate the performance of the proposed approaches in four scenarios, we investigate the following research questions:

RQ$_1$: *How effective is the proposed unsupervised COOD when compared to unsupervised baselines?*

RQ$_2$: *How effective is the proposed weakly-supervised COOD+ when compared to weakly-supervised baselines?*

RQ$_3$: *How effective is the proposed weakly-supervised COOD+ when using different modules or encode backbone?*

RQ$_4$: *Is the main task (Code Search) performance affected by our COOD/COOD+ auxiliary, and to what extent?*

5.4.1 Datasets

In our experiments, we rely on two benchmark datasets: CodeSearchNet (CSN) [81] and TLCS [188]. CSN contains bimodal data points consisting of code paired with function-level NL descriptions (*i.e.,* first lines of documentation comments) in six PLs (*e.g.,* Python, Java) collected from GitHub repositories. While CSN was originally created for a specific downstream task (*i.e.,* code search), it has since been widely adopted by large (NL, PL) models [80, 81] for pre-training due to the informative nature of bimodal instances. Large NL-PL models are first pre-trained across *all* six languages, and then further fine-tuned for a *specific* PL for some downstream task to enhance performance. For code search, the goal is to retrieve the most relevant code given a NL query, where CSN is widely used to further fine-tune a PL-specific code search model [72].

Salza *et al.* [188] used training samples from CSN for pre-training, and created a new dataset sourced from StackOverflow (SO) for fine-tuning the code search model, involving only *three* PLs: Java, Python, and JavaScript. Specifically, they leverage SO user questions as search queries and accepted answers as retrieved code snippets, which differ from GitHub comments and the corresponding code in CSN. We refer to this new dataset as TLCS. Existing work [15, 157, 234] investigated code clones between SO and GitHub, demonstrating there exists *only* 1-3% code reuse. Besides code, user questions in SO

are typically formulated before code answers, without concrete knowledge of what code answers will be, and are mostly written by end-users. Conversely, in GitHub, method docstrings (*i.e.,* comments) are often written following code snippets, and are mostly written by developers. These distinctions cause performance shortfall when directly applying models trained on CSN to TLCS without further fine-tuning or transfer learning [80, 105, 188].

5.4.2 OOD Scenarios

We design four distinct OOD scenarios using the datasets described above, with CSN-Java and CSN-Python as inliers due to their common use for the pre-training of code models.

Scenario 1: Out-domain. Following existing ML work [94, 171, 252], we create an out-of-domain setting by choosing OOD samples from a different dataset than the training data. Thus, samples from TLCS-Java or TLCS-Python are treated as outliers accordingly. Inliers and their corresponding outliers belong to the same PL to ensure approaches don't identify OODs based on syntax differences between PLs but on data domains: GitHub vs. SO. Prior studies [210] show that CSN queries are longer than SO questions on average, so we sampled TLCS questions and answers to match the length distribution of CSN comments and code, to avoid OOD approaches exploiting spurious cues of query length differences. We didn't consider other code search datasets [144, 178, 238] because they either contain only one of the PLs (Python or Java) or have a smaller dataset size.

Scenario 2: Misaligned. In this scenario, we shuffle normal NL-PL pairs so that each code doesn't match its NL description. Although the NL modality sourced from attached comments in code are typically aligned with the PL modality, documentation errors may still occur and not effectively filtered by handcrafted rules [81].

Scenario 3: Shuffled-comment. For (comment, code) pairs, we modify the syntactic information in each comment by shuffling 20% of selected tokens using a seeded random algorithm [193] with positions of stopwords and punctuations unchanged. No changes are made to the code for this scenario. This scenario is inspired by [160, 194]. [194] discovered that NL pre-trained models are insensitive to permuted sentences, which contrasts with

human behavior as humans struggle to understand ungrammatical sentences, or interpret a completely different meaning from a few changes in word order. [160] further introduces syntactic (shuffling) outliers into NL pre-training corpora to enhance OOD robustness and NL understanding performance.

Scenario 4: Buggy-code. We create buggy code using a *semantic* mutation algorithm which injects more natural and realistic bugs into code than other traditional loose/strict mutators [182]. This simulates buggy programs that the model may encounter during testing, typically absent from the training dataset, and should be taken into account by OOD code detectors according to the OOD definition [89]. We avoid using real bug/vulnerability datasets [115, 224, 254] due to limitations like the absence of paired comments, lack of support for Python or Java, introduction to a new dataset domain *etc.*. We generate buggy code for each code in CSN-Java and CSN-Python using [182] to serve as outliers, ensuring the inliers and outliers are from the same dataset domain with the only difference being normal vs. buggy code. We focus on variable-misuse bugs, as only this mutation algorithm is available for both Python and Java in [182]. Variable-misuses occur when a variable name is used but another was meant in scope, and often remain undetected after compilation and regarded as hard-to-detect by recent bug detection techniques [89, 181]. Comments remain unchanged for this scenario.

5.4.3 Model Configurations

For the weakly-supervised COOD+, we experiment with either the contrastive learning module (COOD+_CL) or the binary OOD rejection module (COOD+_BC) to compare against the combined model. All models are trained using the Adam optimizer with a learning rate of $1e-5$, and a linear schedule with 10% warmup steps. The batch size is set to 64, and the number of training epochs is 10. For the COOD+_CL and COOD+, the margins in the margin-based loss are set to 0.2. The balancing value λ is set to 0.2 after a grid search. The hidden layer size in the binary OOD rejection module for COOD+ is 384 (768/2). We also explore the robustness and agnosticism of our COOD+ approach to

different NL-PL models by replacing the GraphCodeBERT encoder with CodeBERT [72], UniXcoder [80], and ContraBERT [150].

Table 5.1: Dataset statistics for weakly-supervised COOD+.

	CodeSearchNet-Java			CodeSearchNet-Python		
	train	valid	test	train	valid	test
ID	142,502	15,838	2,199	217,577	24,178	4,281
Out-domain	1,484	164	2,191	2,266	251	2,983
Misaligned	1,483	163	2,191	2,265	251	2,983
Shuffled-comment	1,478	163	2,183	2,264	251	2,978
Buggy-code	1,484	164	2,191	2,266	251	1,693

5.4.4 OOD Detection Model Training and Measurement

For unsupervised COOD, we use only ID data for model training, thus involving all training data from CSN-Python and CSN-Java, with 10% randomly sampled for validation. We avoid using the CSN development dataset for validation due to its smaller size. For weakly-supervised COOD+, we randomly select 1% of the training data and replaced them with OOD samples generated for each scenario (following [205]), resulting in a total of 4% OOD samples and 96% ID samples for training. During inference, both COOD and COOD+ utilize the same ratio (20%) for inliers and outliers from each scenario, which is more convincing than using an imbalanced dataset (*i.e.,* tiny number of OOD data). Detailed dataset statistics are provided in the Table 4.1. Since all outliers are randomly selected, we report average OOD detection results across *five* random seeds of the test dataset to ensure evaluation reliability and reproducibility.

Following prior work in ML [96,156], we use two standard metrics to measure the effectiveness of our COOD/COOD+ models: the area under the receiver operating characteristic curve (AUROC) and the false positive rate at 95% (FPR95). AUROC is threshold-independent, calculating the area under the ROC curve over a range of threshold values, representing the trade-off between true positive rate and false positive rate. It quantifies the probability that a positive example (ID sample) receives a higher score than a negative one (OOD sample). Higher AUROC indicates better performance. Additionally, FPR95

corresponds to the false positive rate (FPR) when the true positive rate of ID samples is 95%. FPR95 is threshold-dependent, where OODs are identified by setting a threshold σ with $P^{OOD} < 1 - \sigma$ ($P^{ID} > \sigma$) so that a high fraction (95%) of ID data is above the threshold. It measures the proportion of OOD samples that are mistakenly classified when 95% of ID samples are correctly recalled based on the threshold. Lower FPR95 indicates better performance.

5.4.5 Baselines

We compare our COOD/COOD+ against various OOD detection baselines, including adaptations of existing unsupervised NLP OOD approaches on NL-PL encoders (1-2), weakly-supervised approaches based on outlier exposure (3), and neural bug detection techniques (4-5). Since unsupervised approaches (1-2) rely on classification outputs for OOD scoring, we reformulate code search as binary classification to fine-tune the encoders similarly to [72]. (1-2) is supervised for code search, but unsupervised for OOD detection. For weakly-supervised baselines (3-5), we use the same number of OOD samples as COOD+ for a fair comparison. Note that the encoder backbone of (1-3) is also GraphCodeBERT. (4-5) are specifically designed for neural bug detection, thus not requiring other encoder backbone for OOD detection.

1. **Supervised Contrastive Learning For Classification (SCL)** [122]. This method fine-tunes transformer-based classification models by maximizing similarity of input pairs if they are from the same class and minimize it otherwise. Following [252], we adopts MSP, Energy, and Mahalanobis OOD scoring algorithms for OOD detection.

2. **Margin-based Contrastive Learning for Classification (MCL)** [252]. This approach fine-tunes transformer-based classification models by minimizing the L2 distances between instances from the same class, and encouraging the L2 distances between instances of different classes to exceed a margin. We also detect OODs by applying MSP, Energy, and Mahalanobis OOD scoring algorithms.

3. **Energy-based Outlier Exposure (EOE)** [151]. This approach uses a few auxiliary OOD data to fine-tune the classification model with an energy-based margin loss [151], and then utilize Energy scores for OOD detection.

4. **CuBERT** [119]. CuBERT is pre-trained on a large code corpus using masked language modeling, then fine-tuned for bug detection and repair. We adapt CuBERT for OOD classification by alternatively fine-tuning it on our datasets with comments appended to their corresponding code, as CuBERT only accepts single instance inputs.

5. **2P-CuBERT** [89]. This method enhances CuBERT's bug detection accuracy with a two-phase fine-tuning approach. The first phase utilizes contrastive learning on generated synthetic buggy code [18]. For the second phase, we alternatively fine-tune CuBERT to detect OOD using our datasets. Results are reported only for CSN-Python due to the lack of Java bug generation algorithms in [89].

5.4.6 Main Task Performance Analysis

An effective OOD detector, serving as an auxiliary component, should identify and reject OOD samples without negatively impacting the original model's performance on the main downstream task with ID data [252]. Consequently, we validate the effectiveness of our COOD/COOD+ auxiliary on the code search task using the official evaluation benchmark [81, 150] by calculating the mean reciprocal rank (mRR) for each pair of comment-code data over distractor codes in the testing code corpus. Specifically, we first measure the performance of original GraphCodeBERT code search model on both ID and OOD data, whose performance is expected to be negatively affected with the presence of OOD samples. Then, we utilize our COOD/COOD+ auxiliary to filter the testing dataset by setting a threshold to retain 95% of ID instances with higher scores (following existing ML work [165] and the FPR95 definition), as real-world deployment typically involves few OODs. Finally, we directly use the fine-tuned encoder in COOD/COOD+ to perform code search but on the retained ID instances, and compare this performance with that on the

ground-truth ID instances. If the performance loss is recovered by using COOD/COOD+, we actually enhance the trustworthiness and robustness of the original code search model (as shown in Sec. VI-D). Here trustworthiness and robustness mean that predictions of code models become more reliable when encountering OOD data in real-world deployment. Note that the dataset used for COOD/COOD+ training is the same as that used for PL-specific training of existing SOTA code search models.

Table 5.2: Effectiveness of our COOD and COOD+ models compared with the baselines on the CSN-Python dataset.

Approaches	**Out-domain+ID**		**Misaligned+ID**		**Shuffled-comment+ ID**		**Buggy-code+ ID**		**Overall (All OODs+ID)**	
	AUROC↑	**FPR95↓**	**AUROC↑**	**FPR95↓**	**AUROC↑**	**FPR95↓**	**AUROC↑**	**FPR95↓**	**AUROC↑**	**FPR95↓**
Unsupervised										
SCL+MSP	78.21	68.93	49.80	97.23	60.91	90.36	49.23	95.26	60.79	87.05
SCL+Energy	77.76	70.13	65.07	96.11	61.24	90.04	49.58	95.00	65.09	86.95
SCL+Maha	73.34	82.82	73.21	92.13	68.03	87.02	**53.89**	**92.43**	68.72	88.14
MCL+MSP	81.18	65.57	53.51	96.43	62.17	90.44	48.69	94.42	63.11	85.78
MCL+Energy	82.55	64.03	62.57	95.43	63.03	90.52	**48.26**	94.90	66.02	85.16
MCL+Maha	53.05	94.60	62.47	93.12	49.23	95.73	51.13	93.64	54.31	94.35
COOD	**86.60**	**48.25**	**99.85**	**0.16**	**72.82**	**85.18**	49.17	**93.58**	**80.50**	**52.31**
Weakly-supervised										
EOE	**98.96**	**3.26**	91.18	50.03	**98.37**	**4.07**	94.78	24.69	95.95	20.02
CuBERT	92.56	14.46	91.13	17.33	88.92	21.74	60.93	77.73	86.11	27.38
2P-CuBERT	92.26	15.16	84.42	30.83	86.38	26.92	92.87	13.94	88.51	22.65
COOD+	98.80	4.30	**99.53**	**0.40**	98.02	6.52	**97.90**	**5.96**	**98.64**	**4.09**
COOD+_CL	93.91	25.44	99.89	0.17	82.57	72.97	52.56	93.99	85.83	42.57
COOD+_BC	96.49	8.67	74.27	61.38	97.53	5.68	95.71	10.95	90.43	22.98

Table 5.3: Effectiveness of our COOD and COOD+ models compared with the baselines on the CSN-Java dataset.

Approaches	**Out-domain+ID**		**Misaligned+ID**		**Shuffled-comment+ ID**		**Buggy-code+ ID**		**Overall (All OODs+ID)**	
	AUROC↑	**FPR95↓**	**AUROC↑**	**FPR95↓**	**AUROC↑**	**FPR95↓**	**AUROC↑**	**FPR95↓**	**AUROC↑**	**FPR95↓**
Unsupervised										
SCL+MSP	85.15	63.66	58.93	95.76	58.51	92.59	49.01	95.70	62.91	86.92
SCL+Energy	84.00	66.69	55.27	95.56	60.07	91.82	49.13	95.86	62.12	87.48
SCL+Maha	82.16	73.57	79.20	89.71	64.61	91.68	46.55	97.32	68.13	88.07
MCL+MSP	85.16	65.74	59.12	95.83	59.50	95.73	49.44	95.73	63.31	87.40
MCL+Energy	83.44	68.53	44.85	96.22	59.26	91.79	**49.61**	95.51	59.45	88.01
MCL+Maha	50.36	96.73	67.11	90.61	48.44	96.07	46.68	97.44	53.00	95.21
COOD	**92.27**	**40.14**	**99.41**	**0.39**	**75.78**	**86.88**	48.72	**95.04**	**79.05**	**55.95**
Weakly-supervised										
EOE	**99.49**	**1.59**	88.83	62.12	**98.71**	**3.58**	**92.27**	25.51	94.82	23.22
CuBERT	82.59	62.23	49.38	95.25	50.14	94.93	67.97	94.82	62.53	86.80
COOD+	99.29	2.79	**99.56**	**0.81**	97.05	9.43	91.11	**20.88**	**96.75**	**8.48**
COOD+_CL	93.73	23.76	99.65	0.31	83.24	78.14	50.12	95.53	82.47	47.96
COOD+_BC	98.67	3.92	64.55	78.89	95.35	10.17	94.09	14.40	88.16	26.86

5.5 Experimental Results

5.5.1 RQ1: Unsupervised COOD Performance

In this subsection, we analyze the experimental results to assess the detection performance of our unsupervised COOD model compared with the unsupervised baselines. According to Table 5.2 and 5.3, we can observe that COOD outperforms all unsupervised baselines on both CSN-Python and CSN-Java. Notably, COOD effectively detect *out-domain* and *misaligned OOD* testing samples, while other unsupervised approaches only work for the *out-domain* scenario. This is because COOD effectively captures alignment information within (comment, code) pairs through a multi-modal contrastive learning objective with InfoNCE loss and uses similarity scores between comments and code to detect OODs. Specifically, COOD outputs low similarity scores for the out-domain data from TLCS by additionally considering the knowledge gap difference in (comment, code) pairs between ID and out-domain data. Also, as the misaligned scenario involves misaligned (comment, code) pairs, their similarity scores are naturally low. In contrast, the unsupervised baselines aggregate misaligned information into classification logits and rely on the confidence of the "aligned" class to detect OODs. As previously discussed in Sec. IV-C, the contrastive losses [122, 252] used by them are not as effective for learning alignment information, leading to inferior performance. Additionally, detecting token-level OOD in *shuffled-comment* and *buggy-code* scenarios proves challenging without seeing OOD samples during training, as all unsupervised methods fail to detect these OODs.

5.5.2 RQ2: Weakly-supervised COOD+ Performance

We further investigate the performance of our weakly-supervised COOD+ method against several weakly-supervised baselines on CSN-Python and CSN-Java. Table 5.2 shows that weak supervision on a tiny amount of OOD data enables COOD+ (and EOE) to not only address unsupervised COOD's shortcomings in detecting finer-grained *shuffled-comment* and *buggy-code* OODs, but also enhance performance for the *out-domain* scenario for CSN-

Python. This improvement aligns with previous research [95, 123, 151] which enhances OOD detection by complementing the downstream task objective with an complementary discriminator operating to distinguish IDs from external OODs. While EOE slightly outperforms COOD+ for the *out-domain* and *shuffled-comment* scenarios by utilizing the prediction probabilities from one classification module, our COOD+, which combines the BC and CL modules, delivers consistently high performance across all four scenarios, resulting in superior overall performance. In addition, the BC module can be directly adapted to the overall COOD+ framework without modifying the underlying learning objective, but the outlier exposure-based methods (*e.g.,* EOE) typically require additional engineering (*e.g.,* determining class-probabilistic distributions [95], boundaries for energy scores [151]) to equip ML models with OOD detection abilities. Besides, the bug detection method 2P-CuBERT can reasonably detect OODs, but its performance for the *buggy-code* scenario is negatively impacted by the limited amount of training OOD examples.

On the CSN-Java dataset, our COOD+ also achieves the best overall performance compared to all baselines, despite trailing slightly behind EOE for *out-domain* and *shuffled-comment* OODs. While EOE has higher AUROC score than that of COOD+ for the *buggy-code* scenario, it suffers from a high FPR95, indicating a higher margin of error for OOD inference using a threshold of 95% ID recall. Moreover, similar to CSN-Python, CuBERT fails to detect OODs effectively on CSN-Java either, likely due to the lack of training examples. In summary, the superior performance of our COOD+ model results from the interplay between the CL and BL modules, where contrastive learning captures high-level alignment between NL-PL input pairs that is naturally suitable for *out-domain* and *misaligned* OODs, while the OOD rejection classifier targets lower-level OOD information from *shuffled-comment* and *buggy-code* samples. Furthermore, by utilizing a weakly-supervised contrastive learning objective that jointly optimizes for OOD detection and the code search task, our method enables effective deployment of the code search model in OOD environments, which will be further studied in Sec. VI-D.

Table 5.4: Our COOD+ model with different encoders.

Encoders	CSN-Java		CSN-Python	
	AUROC↑	**FPR95↓**	**AUROC↑**	**FPR95↓**
GraphCodeBERT	**96.75**	**8.48**	**98.64**	**4.09**
CodeBERT	95.42	10.27	98.59	4.20
UniXcoder	95.49	10.63	97.83	5.91
ContraBERT	96.19	9.32	98.25	4.76

5.5.3 RQ3: Weakly-Supervised COOD+ Performance with Different Model Components and Encoder Backbone

In this subsection, we evaluate the effect of using only the CL (COOD+_CL) or the BC module (COOD+_BC) against the proposed combined COOD+ model to illustrate how COOD+ generalizes in four OOD scenarios. As shown in Table 5.2 and 5.3, COOD+_CL performs well in the *out-domain* and *misaligned* scenarios, which is due to its ability to effectively capture high-level (comment, code) alignment information. COOD+_BC excels in the *out-domain*, *shuffled-comment*, and *buggy-code* scenarios, since it can learn lower-level features from these types of OOD samples. While COOD+_BC maintains acceptable OOD detection performance with high AUROC (>90%) and low FPR95 (<25%), the CL module remains crucial for overall performance, since without it the overall performance of COOD+ will drop below the EOE baseline. Moreover, removing the BC module has a more negative impact on the OOD detection as COOD+ loses the ability to capture the necessary lower-level OOD information for detecting *shuffled-comment* and *buggy-code* OODs. Note that the standalone CL module performs better than the unsupervised COOD overall, demonstrating that our proposed modification to the original CL objective enhance OOD detection by leveraging the margin-based loss. Thus, the combined model's superior performance validates our design choices. That is, the combined scoring function (cosine similarities from CL and the prediction probabilities from BC) is thoughtfully designed to leverage the advantage of each module for high detection accuracy.

Moreover, we compare the detection performance of our COOD+ with various un-

derlying NL-PL pre-trained encoder. Specifically, we compare our choice of GraphCodeBERT [81] against other NL-PL encoders from the literature including its predecessor, CodeBERT [72], and more recent ones such as UniXcoder [80] and ContraBERT [150]. As shown in Table 5.4, all encoders perform within a 1-2% difference, indicating that our COOD+ framework is robust across different encoders. This demonstrates our framework's flexibility and effectiveness in detecting OODs when deploying various NL-PL encoders for code-related tasks. Furthermore, we investigate key hyperparameters in COOD+, such as m for margin-based contrastive loss and λ in the overall loss function. The detailed results are available in our online appendix [233].

5.5.4 RQ4: Main Task Performance

We present the code search performance under the impact of OOD instances by using GraphCodeBERT (GCB), COOD/COOD+, and the closest competitor EOE in Table 5.5. As described in Sec. V-F, we use the official metric mRR and follow the same testing scheme as the original GraphCodeBERT code search model for evaluation. From Table 5.5, we first observe that our COOD/COOD+ achieves performance comparable to GraphCodeBERT, while the EOE suffers from a significant reduction in performance, as it reformulates code search as binary classification to gain OOD detection ability. This reveals a critical trade-off between OOD detection and downstream task performance. To further validate the importance of OOD detection for code search, we construct outliers based on the CSN-Java and -Python testing dataset, respectively. Given that code search aims to retrieve the most aligned code from a code corpus given an NL query, the outliers are only sampled from three OOD scenarios: *out-domain*, *shuffled-comment* and *buggy-code*, each replacing 5% ID data of the original testing set. We then show the results when the dataset contains 15% OOD samples (*i.e.,* 15% outliers), discard OOD samples by filtering the testing set by ground-truth labels (*i.e.,* Filtered-GT) or using various OOD detection models (*i.e.,* Filtered-OOD-model). Note that the Filtered-GT dataset is the original CSN's subset with 15% of ID samples removed.

Table 5.5: Code search performance under the impact of OOD detection. Higher numbers represent better performance

Dataset	Testing Subset	GCB	EOE	COOD	COOD+
CSN-Python	Origin	69.20	50.11	68.47	69.69
	15% outliers	65.85	43.68	64.67	65.67
	Filtered-GT	70.24	44.85	68.95	70.24
	Filtered-OOD-model	–	46.82	70.30	**73.10**
CSN-Java	Origin	69.10	46.29	68.85	69.46
	15% outliers	64.99	37.77	64.86	64.54
	Filtered-GT	69.12	38.94	69.36	69.93
	Filtered-OOD-model	–	39.33	71.02	**73.18**

According to Table 5.5, the performance of the original GraphCodeBERT code search model drops by 4.84% and 5.95% ((69.10-64.99)/69.10) mRR when outliers are present in CSN-Python and -Java, respectively. As a solution to this issue, our COOD/COOD+ detector recover the performance losses by identifying and filtering out the OOD samples without negatively impacting the model's code understanding ability in code search. Specifically, the code search performance of COOD/COOD+ on the Filtered-COOD/COOD+ dataset (70.30%/73.10% and 71.02%/73.18% on CSN-Python and -Java, respectively) is comparable to or even better than GraphCodeBERT on the Filtered-GT dataset (70.24% and 69.12% on CSN-Python and -Java, respectively). This slight improvement is probably because our detectors filter out additional lower-quality testing samples that resemble outliers. Thus, our COOD/COOD+ detectors enhance the trustworthiness and robustness of the GraphCodeBERT, since the model's predictions become more reliable when encountering OOD data. Note that the original GraphCodeBERT is not equipped with the OOD detection ability, so its corresponding cells for the Filtered-OOD-model in Table 5.5 are left blank.

5.6 Discussions

5.6.1 Analysis of the Overconfidence of MSP with Conformal Prediction

Given an OOD testing sample, ML models pre-trained on ID data are prone to predict a higher MSP confidence score than the threshold and incorrectly identify it as an ID sample [141, 146]. This overconfidence issue limits the effectiveness of OOD detection. For NL data, this is caused by the spurious correlation between OOD and ID features such as entities and syntactic structures [226, 251]. Such correlation also occurs in PL data. For example, an OOD PL input with the syntactic structure "def ... if ... return ... else ... return ..." may receive an ID score if this pattern is commonly used in other ID inputs. To overcome overconfident predictions, previous work explored techniques such as temperature scaling [146], confidence calibration using adversarial samples [27, 141], or adaptive class-dependent threshold [226]. In contrast, our proposed COOD+ utilizes a weakly-supervised contrastive learning objective to take advantage of a small number of OOD samples during training and prevent the alignment between OOD pairs. Moreover, we adopt the binary OOD rejection module to discriminate the fused OOD and ID representations. We further verify whether COOD+ overcomes the overconfidence issue through the lens of Conformal Prediction (CP) [19].

Conformal Prediction (CP) involves post-processing uncertainty quantification techniques that are model-agnostic, and provide statistical guarantees on the predictions of a trained model [19]. The commonly used split CP technique first computes the nonconformity scores, which are OOD scores in our case, and class-dependent thresholds on a calibration set independent of the training data. Then, it builds a prediction set for each testing sample $\mathcal{C}_\alpha(t_l^{test}, c_l^{test})$ satisfying the condition $P(y_l^{test} \in \mathcal{C}_\alpha(t_l^{test}, c_l^{test})) \geq 1 - \alpha$, where α is a small error rate (e.g., 0.05) that the user is willing to tolerate. Here, this condition guarantees that the true outcome is covered by the prediction set with probability $1 - \alpha$, which is also known as the CP coverage. When CP is applied to the OOD detection scores, all scores have the same statistical guarantee, but better OOD scores will

Table 5.6: Effectiveness of COOD+ compared to selected methods for overcoming overconfident OOD predictions.

Methods	CSN-Java		CSN-Python	
	Coverage↑	**P-Set Size↓**	**Coverage↑**	**P-Set Size↓**
MCL+MSP	**97.03**	1.891	95.00	1.834
COOD	95.66	1.593	**95.81**	1.576
COOD+	95.31	**1.077**	95.35	**1.010**

give tighter prediction sets. Conversely, worse scores will give large and uninformative prediction sets, which corresponds to ineffective OOD detection caused by overconfident predictions.

In our experiment, we apply split CP by reserving 20% of the testing samples from each testing dataset (CSN-Java and CSN-Python) for CP calibration, and construct the prediction sets with tolerable error rate $\alpha = 0.05$ on the remaining testing samples. To assess how effectively COOD+ addresses the overconfidence issue, we compare its average prediction set (P-Set) size (between 1 and 2 for binary predictions) with that of selected baselines including MCL+MSP, the best performing approach using MSP OOD scores, and our proposed COOD. As observed in Table 5.6, the proposed COOD+ achieves the smallest prediction sets on both datasets. Specifically, the vast majority of prediction sets obtained by COOD+ contain one value that is 95% statistically guaranteed to be the true OOD label according to the CP condition, indicating the minimal overconfidence of OOD scores. In contrast, the MCL+MSP method is prone to overconfidence, because it produces large prediction sets (i.e., size 2) including both IDs and overconfident OODs. Additionally, without utilizing OOD samples during training, COOD cannot effectively prevent overconfident predictions. Note that in the CP context, although higher coverage is desired, the main goal is to build the smallest prediction sets given the user-specified error rate of 0.05. Therefore, COOD+ is the most effective at overcoming the overconfidence barrier despite its slightly lower coverage than that of MCL+MSP and COOD.

5.6.2 OOD Detection with Large Language Models (LLMs).

It's worth noting that transformer-based code models (*e.g.,* GraphCodeBERT [81]) and LLMs share the same underlying transformer architecture. Scaling up transformer-based code models and training them on vast amounts of code data allows LLMs [183] to perform a wide range of code-related tasks, making coding less labor-intensive and more accessible to end-users. Since LLMs are transformer-based, they are also vulnerable to OOD data, with potentially worse performance degradation due to error accumulation during auto-regressive inference. Thus, identifying OOD samples is crucial to knowing when to trust LLM outputs. Our proposed OOD code framework techniques can be applied to these larger transformer-based code models, similarly as demonstrated in our experiments with different encoders in Table 5.4.

5.6.3 Generalization of COOD/COOD+ to Other Code-related Tasks.

Our COOD/COOD+ framework can be applied for any code-related tasks, particularly code understanding tasks, as long as their input consists of (comment, code) pairs. During software development, developers often write comments following code snippets (methods/functions). Thus, from a realistic perspective, our framework can be generalized to many code understanding tasks, such as clone detection and defect detection, beyond code search. All that is needed is to determine the ID dataset and the out-domain data since all four OOD scenarios are generally relevant to every code understanding task. For instance, in clone detection, before checking whether two (comment, code) pairs are clones, we can first input each pair into our framework (after dataset-specific training) to identify whether they are OODs under the four scenarios designed in our benchmark. Unfortunately, the currently available clone and defect detection datasets only include code without corresponding comments. This is why we haven't yet applied our framework to these tasks. However, there is every reason to believe that our framework will be useful for these tasks when more realistic bi-modal datasets are available in the future.

5.7 Threats to Validity

5.7.1 Construct Validity

Our COOD/COOD+ framework uses data-driven techniques to synthesize OOD samples, which may not fully reflect real-world SE scenarios. While we include diverse OOD scenarios, a pilot study with developers is necessary. Additionally, the reliability of our OOD benchmark depends heavily on the quality of the OOD datasets used.

5.7.2 Internal Validity

Hyperparameter tuning impacts ML performance. For model fine-tuning, we kept the GraphCodeBERT architecture unchanged due to feasibility reasons, but conducted ablation studies with various model components, encoder backbones, and key hyperparameters.

5.7.3 External Validity

We conduct OOD detection experiments on two large-scale code search datasets. Although our focus on Python and Java limits generalizability, experiments on these two languages partially demonstrate that our approach is PL-agnostic.

5.8 Chapter Summary

We proposed two multi-modal OOD detection aproaches for code-related pre-trained ML models; namely unsupervised COOD and weakly-supervised COOD+. The COOD merely leveraged unsupervised contrastive learning to identify OOD samples. As an extension of COOD, COOD+ combined contrastive learning and a binary classifier for OOD detection using a small number of labelled OOD samples. To reap the benefits of these two modules, we also devised a novel scoring metric to fuse their prediction results. The evaluation results demonstrated that the integration of the rejection network and contrastive learning can achieve superior performance in detecting all four OOD scenarios for multi-modal NL-

PL data. Additionally, our models can be applied to the downstream SE task, achieving comparable performance to existing code-related models.

Chapter 6

Conclusion

This dissertation advocates for a paradigm shift in software engineering research and practice toward embracing multimodal learning approaches that reflect the inherent diversity of software artifacts. By integrating multiple data modalities, including source code, natural language documentation, GUI visuals, and structural dependencies, this work demonstrates that software engineering tools can achieve significantly greater accuracy, robustness, and real-world applicability. This dissertation introduces and evaluates three core systems that exemplify the potential of multimodal learning in advancing key software engineering tasks.

First, Janus addresses the increasingly prevalent challenge of detecting duplicate video-based bug reports in GUI-intensive mobile applications. By combining visual representation learning, information retrieval, and adaptive frame alignment, Janus effectively captures the visual, textual, and sequential patterns present in bug report videos. Evaluated on a large benchmark comprising 7,290 duplicate detection tasks from 270 real-world bug reports, Janus significantly outperforms existing techniques. Ablation and qualitative analyses further highlight its ability to produce interpretable hierarchical GUI representations and to accurately localize relevant textual content.

Second, Athena tackles the critical task of change impact analysis. This technique combines conceptual coupling, derived from deep semantic code embeddings, with struc-

tural dependency information from program dependence graphs to enable precise impact predictions, without relying on execution traces or historical change data. Evaluated on a newly constructed benchmark of fine-grained bug-fixing commits, ATHENA achieves notable improvements over strong baselines: +10.34% in mean reciprocal rank, +9.55% in mean average precision, and +11.68% in HIT@10. Its strength is particularly evident when identifying impacted methods outside the query method's class, illustrating its capacity to model long-range semantic and structural relationships.

Third, this dissertation extends its contributions to address a broader challenge in model reliability: specifically, the detection of OOD inputs in open-world settings. As deep code models are increasingly deployed in practical development environments, the ability to detect anomalous or unseen inputs becomes essential. To meet this need, we introduce COOD and COOD+, the first multimodal OOD detection frameworks tailored for code-related tasks. These techniques apply contrastive learning across both code and natural language modalities and include OOD rejection modules that significantly mitigate performance degradation in the downstream code search task.

Beyond the development of these individual systems, this work represents a broader opportunity to rethink how we model and reason about software. By viewing code, text, visuals, and structural relationships as interconnected perspectives rather than isolated data sources, multimodal learning opens the door to more context-aware, interpretable, and trustworthy software tools. In addition to technical contributions, this dissertation introduces reusable benchmarks and methodologies that offer valuable infrastructure for future research in this area. Looking forward, this research lays the foundation for expanding the reach of multimodal learning in software engineering. Future directions include refining the proposed techniques, extending OOD detection for broader software contexts, and exploring new applications such as code generation, code summarization, and merge conflict resolution. Ultimately, the long-term vision is to enable a new generation of intelligent, adaptive, and reliable software engineering tools that support developers throughout the full lifecycle of modern software systems.

Bibliography

[1] Cisco systems https://www.cisco.com.

[2] Tesseract ocr library https://github.com/tesseract-ocr/tesseract/wiki.

[3] Droidweight https://test.f-droid.org/de/packages/de.delusions.measure/index.html, 2020.

[4] Gnucash https://github.com/codinguser/gnucash-android, 2020.

[5] Android apps for screen recording: https://www.androidauthority.com/best-screen-recording-apps-600838/, 2023.

[6] Android screenshot and video recording features: https://support.google.com/android/answer/9075928?hl=en, 2023.

[7] Checkmarx, 2023.

[8] Codeql, 2023.

[9] Fdroid https://f-droid.org/en/, 2023.

[10] Firefox focus https://github.com/mozilla-mobile/focus-android, 2023.

[11] Github video uploads: https://github.blog/2021-05-13-video-uploads-available-github/, 2023.

[12] Gpstest https://github.com/barbeau/gpstest, 2023.

[13] Images to pdf https://github.com/Swati4star/Images-to-PDF, 2023.

[14] Lucene's tfidf similarity javadoc - https://tinyurl.com/ybhqqrqm, 2023.

[15] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. On code reuse from stackoverflow: An exploratory study on android apps. *Inf. Softw. Technol.*, 88:148–158, 2017.

[16] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 746–755, New York, NY, USA, 2011. Association for Computing Machinery.

[17] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei 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*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.

[18] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *NeurIPS*, 34:27865–27876, 2021.

[19] Anastasios N Angelopoulos, Stephen Bates, et al. Conformal prediction: A gentle introduction. *Found. Trends Mach. Learn.*, 16(4):494–591, 2023.

[20] Robert S Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[21] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 9 pp.–, 2005.

[22] Sean Banerjee, Zahid Syed, Jordan Helmick, and Bojan Cukic. A fusion approach for classifying duplicate problem reports. In *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, November 2013.

[23] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers. *arXiv preprint arXiv:2106.08254*, 2021.

[24] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, June 2018.

[25] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, June 2020.

[26] Carlos Bernal-Cárdenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. Guigle: a gui search engine for android apps. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, page 71–74. IEEE Press, 2019.

[27] Julian Bitterwolf, Alexander Meinke, and Matthias Hein. Certifiably adversarially robust detection of out-of-distribution data. *NeurIPS*, 33:16085–16095, 2020.

[28] Markus Borg, Krzysztof Wnuk, Björn Regnell, and Per Runeson. Supporting change impact analysis using a recommendation system: An industrial case study in a safety-critical context. *IEEE Transactions on Software Engineering*, 43(07):675–700, jul 2017.

[29] Ben Breech, Anthony Danalis, Stacey Shindo, and Lori Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of the IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 453–457, 2004.

[30] Ben Breech, Mike Tegtmeyer, and Lori Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 55–65, 2006.

[31] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[32] Max Brunsfeld, Patrick Thomson, Andrew Hlynskyi, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Hendrik van Antwerpen, Michael Davis, Ika, Tuan-Anh Nguyen, Stafford Brunk, Niranjan Hasabnis, bfredl, Mingkai Dong, Vladimir Panteleev, ikrima, Steven Kalt, Kolja Lampe, Alex Pinkus, Mark Schmitz, Matthew Krupcale, narpfel, Santos Gallegos, Vicent Martí, Edgar, and George Fraser. tree-sitter/tree-sitter: v0.20.7, September 2022.

[33] Haipeng Cai. A reflection on the predictive accuracy of dynamic impact analysis. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566, 2020.

[34] Haipeng Cai and Raul Santelices. Diver: precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 343–348, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Haipeng Cai and Raul Santelices. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *Journal of Systems and Software*, 103:248–265, 2015.

[36] Haipeng Cai, Raul Santelices, and Siyuan Jiang. Prioritizing change-impact analysis via semantic program-dependence quantification. *IEEE Transactions on Reliability*, 65(3):1114–1132, 2016.

[37] Haipeng Cai, Raúl A. Santelices, and Douglas Thain. Diapro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *ACM Trans. Softw. Eng. Methodol.*, 25(2):18:1–18:50, 2016.

[38] Haipeng Cai and Douglas Thain. Distia: a cost-effective dynamic impact analysis for distributed programs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 344–355, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[40] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, and W. Liu. Coca: Improving and explaining graph neural network-based vulnerability detection systems. In *ICSE*, pages 939–939, 2024.

[41] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments. *Advances in neural information processing systems*, 33:9912–9924, 2020.

[42] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9650–9660, 2021.

[43] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 86–96, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. Reformulating queries for duplicate bug report detection. In *Proceedings of the IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, pages 218–229, 2019.

[45] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. From lost to found: Discover missing ui design semantics through reovering missing tags. In *Proceedings of the ACM on Human-Computer Interaction*, volume 4, pages 1–22. Association for Computing Machinery (ACM), October 2020.

[46] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From UI design image to GUI skeleton. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, May 2018.

[47] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. Wireframe-based UI design search through image autoencoder. *ACM Transactions on Software Engineering and Methodology*, 29(3):1–31, June 2020.

[48] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. Extracting replayable interactions from videos of mobile app usage. *arXiv preprint arXiv:2207.04165*, 2022.

[49] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, November 2020.

[50] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[51] Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. Plur: A unifying, graph-based view of program learning, understanding, and repair. *NeurIPS*, 34:23089–23101, 2021.

[52] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi, editors, pages 103–111, Doha, Qatar, October 2014. Association for Computational Linguistics.

[53] Chien-Li Chou, Hua-Tsung Chen, and Suh-Yin Lee. Pattern-based near-duplicate video retrieval and localization on web-scale videos. *IEEE Transactions on Multimedia*, 17:382–395, 2015.

[54] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source

code: A tool demonstration. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, pages 516–519, 2013.

[55] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 957–969. IEEE, 2021.

[56] Jose Luis de la Vara, Markus Borg, Krzysztof Wnuk, and Leon Moonen. An industrial survey of safety evidence change impact analysis practice. *IEEE Transactions on Software Engineering*, 42(12):1095–1117, 2016.

[57] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 3844–3852, Red Hook, NY, USA, 2016. Curran Associates Inc.

[58] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pages 845–854, 2017.

[59] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

[60] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[61] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Softw. Engg.*, 18(2):277–309, apr 2013.

[62] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[63] Mingzhe Du, Shengcheng Yu, Chunrong Fang, Tongyu Li, Heyuan Zhang, and Zhenyu Chen. Semcluster: a semi-supervised clustering tool for

crowdsourced test reports with deep image understanding. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1756–1759, 2022.

[64] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. CenterNet: Keypoint triplets for object detection. In *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, October 2019.

[65] Viet Duong, Qiong Wu, Zhengyi Zhou, Eric Zavesky, Jiahe Chen, Xiangzhou Liu, Wen-Ling Hsu, and Huajie Shao. General-purpose multi-modal ood detection framework. *TMLR*, 2024.

[66] Camilo Escobar-Velásquez, Michael Osorio-Riaño, and Mario Linares-Vásquez. Mutapk: Source-codeless mutant generation for android apps. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1090–1093. IEEE, 2019.

[67] Sepideh Esmaeilpour, Bing Liu, Eric Robertson, and Lei Shu. Zero-shot out-of-distribution detection based on the pretrained model clip. In *AAAI*, 2022.

[68] Amir Farzad and T Aaron Gulliver. Unsupervised log message anomaly detection. *ICT Express*, 6(3):229–237, 2020.

[69] Mattia Fazzini, Kevin Moran, Carlos Bernal-Cárdenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. Enhancing mobile app bug reporting via real-time understanding of reproduction steps. *IEEE Trans. Softw. Eng.*, 49(3):1246–1272, March 2023.

[70] Sidong Feng and Chunyang Chen. Gifdroid: automated replay of visual bug reports for android apps. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1045–1057, 2022.

[71] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. Read it, don't watch it: Captioning bug recordings automatically. *arXiv preprint arXiv:2302.00886*, 2023.

[72] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu, editors, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[73] Stephen Fink and Julian Dolby. Wala–the tj watson libraries for analysis, 2012.

[74] Stanislav Fort, Jie Ren, and Balaji Lakshminarayanan. Exploring the limits of out-of-distribution detection. *NeurIPS*, 34:7068–7081, 2021.

[75] Jingwen Fu, Xiaoyi Zhang, Yuwang Wang, Wenjun Zeng, Sam Yang, and Grayson Hilliard. Understanding mobile gui: from pixel-words to screen-sentences. *arXiv preprint arXiv:2105.11941*, 2021.

[76] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE)*, pages 430–440, 2012.

[77] Otis Gospodnetic, Erik Hatcher, and Douglas R. Cutting. Lucene in action. 2004.

[78] Alex Graves and Alex Graves. Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45, 2012.

[79] Jean-Bastien Grill, Florian Strub, Florent Altch'e, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Ávila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, and Michal Valko. Bootstrap your own latent: A new approach to self-supervised learning. *ArXiv*, abs/2006.07733, 2020.

[80] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and 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)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, pages 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[81] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366, 2020.

[82] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *ESEC/FSE*, page 1387–1397, 2020.

[83] Shir Gur, Ameen Ali, and Lior Wolf. Visualization of supervised and self-supervised neural networks via attribution guided factorization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13):11545–11554, May 2021.

[84] Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*,

ISSTA 2017, page 318–328, New York, NY, USA, 2017. Association for Computing Machinery.

[85] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *CVPR*, volume 2, pages 1735–1742, 2006.

[86] Hossein Hajipour, Ning Yu, Cristian-Alexandru Staicu, and Mario Fritz. Simscood: Systematic analysis of out-of-distribution generalization in fine-tuned source code models. In *NAACL*, pages 1400–1416, 2024.

[87] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011.

[88] Rui Hao, Yang Feng, James A Jones, Yuying Li, and Zhenyu Chen. Ctras: Crowdsourced test report aggregation and summarization. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 900–911. IEEE, 2019.

[89] Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. On distribution shift in learning-based bug detectors. In *ICML*, pages 8559–8580. PMLR, 2022.

[90] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

[91] Matthias Hein, Maksym Andriushchenko, and Julian Bitterwolf. Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In *CVPR*, pages 41–50, 2019.

[92] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv:1610.02136*, 2016.

[93] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *ICLR*, 2017.

[94] Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzic, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. *arXiv:2004.06100*, 2020.

[95] Dan Hendrycks, Mantas Mazeika, and Thomas Dietterich. Deep anomaly detection with outlier exposure. *arXiv:1812.04606*, 2018.

[96] Dan Hendrycks, Mantas Mazeika, Saurav Kadavath, and Dawn Song. Using self-supervised learning can improve model robustness and uncertainty. *NeurIPS*, 32, 2019.

[97] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim

Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristóf Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matús Sulír, Fatemeh H. Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling. *CoRR*, abs/2011.06244, 2020.

[98] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21, 04 2015.

[99] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013.

[100] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531, 2015.

[101] MD Zakir Hossain, Ferdous Sohel, Mohd Fairuz Shiratuddin, and Hamid Laga. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys (CsUR)*, 51(6):1–36, 2019.

[102] Yen-Chang Hsu, Yilin Shen, Hongxia Jin, and Zsolt Kira. Generalized odin: Detecting out-of-distribution image without learning from out-of-distribution data. In *CVPR*, pages 10951–10960, 2020.

[103] Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Mike Papadakis, Lei Ma, and Yves Le Traon. Codes: towards code model generalization under distribution shift. In *ICSE-NIER*, pages 1–6. IEEE, 2023.

[104] Yibo Hu and Latifur Khan. Uncertainty-aware reliable text classification. In *KDD*, page 628–636, 2021.

[105] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. Cosqa: 20,000+ web queries for code search and question answering. *arXiv:2105.13239*, 2021.

[106] Zheng Huang, Kai Chen, Jianhua He, Xiang Bai, Dimosthenis Karatzas, Shijian Lu, and CV Jawahar. Icdar2019 competition on scanned receipt ocr and information extraction. In *Proceedings of the 2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 1516–1520. IEEE, 2019.

[107] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

[108] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. Contrastive code representation learning. *arXiv:2007.04973*, 2020.

[109] Mohammad-Amin Jashki, Reza Zafarani, and Ebrahim Bagheri. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, page 84–90, New York, NY, USA, 2008. Association for Computing Machinery.

[110] Yu-Gang Jiang, Chong-Wah Ngo, and Jun Yang. Towards optimal bag-of-features for object categorization and semantic video retrieval. In *Proceedings of the 6th ACM international conference on Image and video retrieval*, pages 494–501, 2007.

[111] Zijian Jiang, Ye Wang, Hao Zhong, and Na Meng. Automatic method change suggestion to complement multi-entity edits. *Journal of Systems and Software*, 159:110441, 10 2019.

[112] Di Jin, Shuyang Gao, Seokhwan Kim, Yang Liu, and Dilek Hakkani-Tür. Towards textual out-of-domain detection without in-domain labels. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 30:1386–1395, 2022.

[113] Weizhen Jing, Xiushan Nie, C. Cui, Xiaoming Xi, Gongping Yang, and Yilong Yin. Global-view hashing: harnessing global relations in near-duplicate video retrieval. *World Wide Web*, 22:771–789, 2019.

[114] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. An empirical investigation into the reproduction of bug reports for android apps. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 321–322, 2022.

[115] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, pages 437–440, 2014.

[116] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18, 10 2012.

[117] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering*, pages 119–128, 2010.

[118] Amita Kamath, Robin Jia, and Percy Liang. Selective question answering under domain shift. In *ACL*, pages 5684–5696, 2020.

[119] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the International conference on machine learning*, pages 5110–5121. PMLR, 2020.

[120] Li Kang. Automated Duplicate Bug Reports Detection - An Experiment at Axis Communication AB. Master's thesis, 2017.

[121] Dimosthenis Karatzas, Lluís Gómez i Bigorda, Anguelos Nicolaou, Suman K. Ghosh, Andrew D. Bagdanov, M. Iwamura, Jiri Matas, Lukás Neumann, Vijay Ramaseshan Chandrasekhar, Shijian Lu, Faisal Shafait, Seiichi Uchida, and Ernest Valveny. Icdar 2015 competition on robust reading. *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1156–1160, 2015.

[122] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *NeurIPS*, 33:18661–18673, 2020.

[123] Jaeyoung Kim, Kyuheon Jung, Dongbin Na, Sion Jang, Eunbin Park, and Sungchul Choi. Pseudo outlier exposure for out-of-distribution detection using pretrained transformers. In *ACL*, pages 1469–1482, 2023.

[124] Jaeyoung Kim, Seo Taek Kong, Dongbin Na, and Kyu-Hwan Jung. Key feature replacement of in-distribution samples for out-of-distribution detection. In *AAAI*, volume 37, pages 8246–8254, 2023.

[125] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[126] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun, editors, 2015.

[127] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[128] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 262–265, New York, NY, USA, 2014. Association for Computing Machinery.

[129] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Splitting commits via past code changes. In *Proceedings of the 2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 129–136, 2016.

[130] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 803–814, New York, NY, USA, 2014. Association for Computing Machinery.

[131] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *Proceedings of the 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*, pages 491–507. Springer, 2020.

[132] Giorgos Kordopatis-Zilos, Symeon Papadopoulos, I. Patras, and Yiannis Kompatsiaris. Near-duplicate video retrieval by aggregating intermediate cnn layers. In *Proceedings of the International Conference on Multimedia Modeling*, 2017.

[133] Giorgos Kordopatis-Zilos, Symeon Papadopoulos, I. Patras, and Yiannis Kompatsiaris. Near-duplicate video retrieval with deep metric learning. *Proceedings of the 2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*, pages 347–356, 2017.

[134] Giorgos Kordopatis-Zilos, Symeon Papadopoulos, Ioannis Patras, and Ioannis Kompatsiaris. Fivr: Fine-grained incident video retrieval. *IEEE Transactions on Multimedia*, 21(10):2638–2652, 2019.

[135] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[136] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Lv Jian, and Alexander Egyed. Do data dependencies in source code complement call dependencies for understanding requirements traceability? In *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 181–190, 2012.

[137] Hiroki Kuramoto, Masanari Kondo, Yutaro Kashiwa, Yuta Ishimoto, Kaze Shindo, Yasutaka Kamei, and Naoyasu Ubayashi. Do visual issue reports help developers fix bugs? a preliminary study of using videos and images to report issues on github. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 511–515, 2022.

[138] Quoc V. Le and Tomás Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[139] Van-Hoang Le and Hongyu Zhang. Log-based anomaly detection with deep learning: How far are we? In *ICSE*, pages 1356–1367, 2022.

[140] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, Yongqiang Yang, and Michael R. Lyu. Heterogeneous anomaly detection for software systems via semi-supervised cross-modal attention. In *ICSE*, page 1724–1736, 2023.

[141] Kimin Lee, Honglak Lee, Kibok Lee, and Jinwoo Shin. Training confidence-calibrated classifiers for detecting out-of-distribution samples. *arXiv:1711.09325*, 2017.

[142] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23, 12 2013.

[143] Minghao Li, Tengchao Lv, Jingye Chen, Lei Cui, Yijuan Lu, Dinei Florencio, Cha Zhang, Zhoujun Li, and Furu Wei. Trocr: Transformer-based optical character recognition with pre-trained models. *arXiv preprint arXiv:2109.10282*, 2021.

[144] Ruitong Li, Gang Hu, and Min Peng. Hierarchical embedding for code search in software q&a sites. In *IJCNN*, pages 1–10. IEEE, 2020.

[145] Xinzhe Li, Ming Liu, Shang Gao, and Wray Buntine. A survey on out-of-distribution evaluation of neural nlp models. In *IJCAI*, pages 6683–6691, 2023.

[146] Shiyu Liang, Yixuan Li, and Rayadurgam Srikant. Enhancing the reliability of out-of-distribution image detection in neural networks. *arXiv:1706.02690*, 2017.

[147] Meng-Jie Lin, Cheng-Zen Yang, Chao-Yuan Lee, and Chun-Chang Chen. Enhancements for duplication detection in bug reports with manifold correlation features. *Journal of Systems and Software*, 121:223–233, 2016.

[148] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 111–122. IEEE Press, 2015.

[149] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *ICSE-SEIP*, pages 338–347, 2021.

[150] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. Contrabert: Enhancing code pre-trained models via contrastive learning. In *ICSE*, page 2476–2487, 2023.

[151] Weitang Liu, Xiaoyun Wang, John Owens, and Yixuan Li. Energy-based out-of-distribution detection. *NeurIPS*, 33:21464–21475, 2020.

[152] Xiaofeng Liu, Chaehwa Yoo, Fangxu Xing, Hyejin Oh, Georges El Fakhri, Je-Won Kang, Jonghye Woo, et al. Deep unsupervised domain adaptation: A review of recent advances and perspectives. *APSIPA Trans. Signal Inf. Proc.*, 11(1), 2022.

[153] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[154] Z. Liu, Z. Tang, J. Zhang, X. Xia, and X. Yang. Pre-training by predicting program dependencies for vulnerability analysis tasks. In *ICSE*, pages 935–935, 2024.

[155] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl eyes. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, December 2020.

[156] Philipp Liznerski, Lukas Ruff, Robert A Vandermeulen, Billy Joe Franks, Klaus-Robert Müller, and Marius Kloft. Exposing outlier exposure: What can be learned from few, one, and zero outlier images. *arXiv:2205.11474*, 2022.

[157] Adriaan Lotter, Sherlock A. Licorish, Bastin Tony Roy Savarimuthu, and Sarah Meldrum. Code reuse in stack overflow and popular open source java projects. In *ASWEC*, pages 141–150, 2018.

[158] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[159] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. Detecting anomaly in big data system logs using convolutional neural network. In *DASC/PiCom/DataCom/CyberSciTech 2018*, pages 151–158. IEEE, 2018.

[160] Kimberly T Mai, Toby Davies, and Lewis D Griffin. Self-supervised losses for one-class textual anomaly detection. *arXiv:2204.05695*, 2022.

[161] Snehashis Majhi, Srijan Das, François Brémond, Ratnakar Dash, and Pankaj Kumar Sa. Weakly-supervised joint anomaly detection and classification. In *FG*, pages 1–7. IEEE, 2021.

[162] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele

Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347, 2021.

[163] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun, editors, 2013.

[164] Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota, and Sonia Haiduc. On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering*, 25, 09 2020.

[165] Yifei Ming, Ziyang Cai, Jiuxiang Gu, Yiyou Sun, Wei Li, and Yixuan Li. Delving into out-of-distribution detection with vision-language representations. *NeurIPS*, 35:35087–35102, 2022.

[166] Kevin Moran, Carlos Bernal-Cardenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, 46(2):196–221, February 2020.

[167] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Christopher Vendome, and Denys Poshyvanyk. Automatically Discovering, Reporting and Reproducing Android Application Crashes . In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 33–44, Los Alamitos, CA, USA, April 2016. IEEE Computer Society.

[168] Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *Proceedings of the 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 138–147, 2013.

[169] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with REMAUI (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, November 2015.

[170] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv:1807.03748*, 2018.

[171] Alexander Podolskiy, Dmitry Lipin, Andrey Bout, Ekaterina Artemova, and Irina Piontkovskaya. Revisiting mahalanobis distance for transformer-based out-of-domain detection. In *AAAI*, volume 35, pages 13675–13682, 2021.

[172] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14:5–32, 02 2009.

[173] Denys Poshyvanyk, Maksym Petrenko, Andrian Marcus, Xinrong Xie, and Dapeng Liu. Source code exploration with google. In *Proceedings of the 2006 22nd IEEE International Conference on Software Maintenance*, pages 334–338, 2006.

[174] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. RoScript. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, June 2020.

[175] Yuning Qiu, Teruhisa Misu, and Carlos Busso. Unsupervised scalable multimodal driving anomaly detection. *IEEE TIV*, 2022.

[176] 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 *Proceedings of the International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[177] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[178] Nikitha Rao, Chetan Bansal, and Joe Guan. Search4code: Code search intent classification using weak supervision. In *MSR*, pages 575–579, 2021.

[179] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[180] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-CNN: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, June 2017.

[181] C. Richter and H. Wehrheim. How to train your neural bug detector: Artificial vs real bugs. In *ASE*, pages 1036–1048. IEEE Computer Society, 2023.

[182] Cedric Richter and Heike Wehrheim. Learning realistic mutations: Bug creation for neural bug detectors. In *ICST*, pages 162–173, 2022.

[183] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv:2308.12950*, 2023.

[184] Lukas Ruff, Robert A. Vandermeulen, Nico Görnitz, Alexander Binder, Emmanuel Müller, Klaus-Robert Müller, and Marius Kloft. Deep semi-supervised anomaly detection. In *ICLR*, 2020.

[185] Sable Research Group. Soot: A java bytecode optimization framework. https://soot-oss.github.io/soot/, 2023.

[186] Mohammadreza Salehi, Hossein Mirzaei, Dan Hendrycks, Yixuan Li, Mohammad Hossein Rohban, and Mohammad Sabokrou. A unified survey on anomaly, novelty, open-set, and out of-distribution detection: Solutions and future challenges. *TMLR*, 2022.

[187] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., USA, 1986.

[188] Pasquale Salza, Christoph Schwizer, Jian Gu, and Harald C Gall. On the effectiveness of transfer learning for code search. *TSE*, 2022.

[189] Adriana Sejfia, Satyaki Das, Saad Shafiq, and Nenad Medvidović. Toward improved deep learning-based vulnerability detection. In *ICSE*, pages 1–12, 2024.

[190] Mark Sherriff and Laurie Williams. Empirical software change impact analysis using singular value decomposition. In *Proceedings of the 2008 1st International Conference on Software Testing, Verification, and Validation*, pages 268–277, 2008.

[191] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. Cocosoda: Effective contrastive learning for code search. In *ICSE*, pages 2198–2210, 2023.

[192] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[193] Koustuv Sinha, Robin Jia, Dieuwke Hupkes, Joelle Pineau, Adina Williams, and Douwe Kiela. Masked language modeling and the distributional hypothesis: Order word matters pre-training for little. In *EMNLP*, pages 2888–2913, 2021.

[194] Koustuv Sinha, Prasanna Parthasarathi, Joelle Pineau, and Adina Williams. Unnatural language inference. *arXiv:2101.00010*, 2020.

[195] R. Smith. An overview of the tesseract OCR engine. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2*. IEEE, September 2007.

[196] Ray Smith. An overview of the tesseract ocr engine. In *Proceedings of the Ninth international conference on document analysis and recognition (ICDAR 2007)*, volume 2, pages 629–633. IEEE, 2007.

[197] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. Toward interactive bug reporting for (android app) end-users. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 344–356, New York, NY, USA, 2022. Association for Computing Machinery.

[198] Sargur N Srihari, Ajay Shekhawat, and Stephen W Lam. Optical character recognition (ocr). In *Encyclopedia of Computer Science*, pages 1326–1333. 2003.

[199] Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *ICSE*, pages 1–13, 2024.

[200] Lei Sun, Kailun Yang, Xinxin Hu, Weijian Hu, and Kaiwei Wang. Real-time fusion network for rgb-d semantic segmentation incorporating unexpected obstacle detection for road-driving images. *IEEE Robot. Autom. Lett.*, 5(4):5558–5565, 2020.

[201] Yiyou Sun, Yifei Ming, Xiaojin Zhu, and Yixuan Li. Out-of-distribution detection with deep nearest neighbors. In *ICML*, 2022.

[202] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, 2014.

[203] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *ArXiv*, abs/1602.07261, 2016.

[204] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes? an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[205] Yu Tian, Gabriel Maicas, Leonardo Zorron Cheng Tao Pu, Rajvinder Singh, Johan W Verjans, and Gustavo Carneiro. Few-shot anomaly detection for polyp frames from colonoscopy. In *MICCAI*, pages 274–284. Springer, 2020.

[206] Marco Torchiano and Filippo Ricca. Impact analysis by means of unstructured knowledge in the context of bug repositories. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, New York, NY, USA, 2010. Association for Computing Machinery.

[207] Antonio Torralba and Alexei A Efros. Unbiased look at dataset bias. In *CVPR*, pages 1521–1528. IEEE, 2011.

[208] Neeraj Varshney, Swaroop Mishra, and Chitta Baral. Investigating selective prediction approaches across several tasks in IID, OOD, and adversarial settings. In *ACL*, pages 1995–2002, 2022.

[209] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, volume 30. Curran Associates, Inc., 2017.

[210] Chaozheng Wang, Zhenhao Nong, Cuiyun Gao, Zongjie Li, Jichuan Zeng, Zhenchang Xing, and Yang Liu. Enriching query semantics for code search with reinforcement learning. *Neural Netw.*, 145:22–32, 2022.

[211] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Information and Software Technology*, 110:139–155, June 2019.

[212] Kai Wang, Boris Babenko, and Serge J. Belongie. End-to-end scene text recognition. In *Proceedings of the 2011 International Conference on Computer Vision*, pages 1457–1464, 2011.

[213] LeiChen Wang, Simon Giebenhain, Carsten Anklam, and Bastian Goldluecke. Radar ghost target detection via multimodal transformers. *IEEE Robot. Autom. Lett.*, 6(4):7758–7765, 2021.

[214] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. Cora: Decomposing and describing tangled code changes for reviewer. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1050–1061, 2019.

[215] Wei Wang, Yun He, Tong Li, Jiajun Zhu, and Jinzhuo Liu. An integrated model for information retrieval based change impact analysis. *Scientific Programming*, 2018:1–13, 03 2018.

[216] Xin Wang, Yasheng Wang, Pingyi Zhou, Fei Mi, Meng Xiao, Yadao Wang, Li Li, Xiao Liu, Hao Wu, Jin Liu, and Xin Jiang. CLSEBERT: contrastive learning for syntax enhanced code pre-trained model. *CoRR*, abs/2108.04556, 2021.

[217] Xuheng Wang, Jiaxing Song, Xu Zhang, Junshu Tang, Weihe Gao, and Qingwei Lin. Logonline: A semi-supervised log-based anomaly detector aided with online learning mechanism. In *ASE*, pages 141–152. IEEE, 2023.

[218] Yue Wang, Weishi Wang, Shafiq Joty, and 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*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[219] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Trans. Softw. Eng. Methodol.*, 31(2), March 2022.

[220] Tyler Wendland, Jingyang Sun, Junayed Mahmud, SM Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. Andror2: A dataset of manually-reproduced bug reports for android apps. In *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 600–604. IEEE, 2021.

[221] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code. *arXiv:2305.04106*, 2023.

[222] Martin White, Michele Tufano, Matías Martínez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490, 2019.

[223] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 87–98, New York, NY, USA, 2016. Association for Computing Machinery.

[224] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *ESEC/FSE*, pages 1556–1560, 2020.

[225] Xiao Wu, Alexander Hauptmann, and Chong-Wah Ngo. Practical elimination of near-duplicates from web video search. *Proceedings of the 15th ACM international conference on Multimedia*, 2007.

[226] Yanan Wu, Keqing He, Yuanmeng Yan, QiXiang Gao, Zhiyuan Zeng, Fujia Zheng, Lulu Zhao, Huixing Jiang, Wei Wu, and Weiran Xu. Revisit overconfidence for ood detection: Reassigned contrastive learning with adaptive class-dependent threshold. In *NAACL-HLT*, pages 4165–4179, 2022.

[227] Zhipeng Wu and Kiyoharu Aizawa. Self-similarity-based partial near-duplicate video retrieval and alignment. *International Journal of Multimedia Information Retrieval*, 3:1–14, 2014.

[228] Ji Xin, Raphael Tang, Yaoliang Yu, and Jimmy Lin. The art of abstention: Selective prediction and error regularization for natural language processing. In *ACL-IJCNLP*, pages 1040–1051, 2021.

[229] Keyang Xu, Tongzheng Ren, Shikun Zhang, Yihao Feng, and Caiming Xiong. Unsupervised out-of-domain detection via pre-trained transformers. In *ACL*, pages 1052–1061, 2021.

[230] Hui Xue, Qiang Yang, and Songcan Chen. Svm: Support vector machines. In *The top ten algorithms in data mining*, pages 51–74. Chapman and Hall/CRC, 2009.

[231] Yanfu Yan, Nathan Cooper, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. Janus replication package: `https://doi.org/10.5281/zenodo.10455811`, 2023.

[232] Yanfu Yan, Nathan Cooper, Kevin Moran, Gabriele Bavota, Denys Poshyvanyk, and Steve Rich. Enhancing code understanding for impact analysis by combining transformers and program dependence graphs. *Proc. ACM Softw. Eng.*, (FSE), 2024.

[233] Yanfu Yan, Viet Duong, Huajie Shao, and Denys Poshyvanyk. Cood online appendix: `https://github.com/yanyanfu/COOD`, 2024.

[234] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. Stack overflow in github: Any snippets there? In *MSR*, pages 280–290, 2017.

[235] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. Generalized out-of-distribution detection: A survey. *IJCV*, 2024.

[236] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *ICSE*, pages 1448–1460. IEEE, 2021.

[237] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. *XLNet: generalized autoregressive pretraining for language understanding*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[238] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *WWW*, pages 1693–1703, 2018.

[239] Jaemin Yoo, Tiancheng Zhao, and Leman Akoglu. Data augmentation is a hyperparameter: Cherry-picked self-supervision for unsupervised anomaly detection is creating the illusion of success. *TMLR*, 2022.

[240] Boxi Yu, Jiayi Yao, Qiuai Fu, Zhiqing Zhong, Haotian Xie, Yaoliang Wu, Yuchi Ma, and Pinjia He. Deep learning or classical machine learning? an empirical study on log-based anomaly detection. In *ICSE*, pages 1–13, 2024.

[241] Shengcheng Yu, Chunrong Fang, Yulei Liu, Ziqian Zhang, Yexiao Yun, Xin Li, and Zhenyu Chen. Universally adaptive cross-platform reinforcement learning testing via gui image understanding. *arXiv preprint arXiv:2208.09116*, 2022.

[242] Zhiyuan Zeng, Hong Xu, Keqing He, Yuanmeng Yan, Sihong Liu, Zijun Liu, and Weiran Xu. Adversarial generative distance-based classifier for robust out-of-domain detection. In *ICASSP*, pages 7658–7662. IEEE, 2021.

[243] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. A transformer-based framework for multivariate time series representation learning. In *KDD*, page 2114–2124, 2021.

[244] Li-Ming Zhan, Haowen Liang, Bo Liu, Lu Fan, Xiao-Ming Wu, and Albert Y.S. Lam. Out-of-scope intent detection with self-supervision and discriminative training. In *ACL*, pages 3521–3532, 2021.

[245] Chenyangguang Zhang, Tong Jia, Guopeng Shen, Pinyan Zhu, and Ying Li. Metalog: Generalizable cross-system anomaly detection from logs with meta-learning. In *ICSE*, pages 938–938. IEEE Computer Society, 2024.

[246] Ting Zhang, DongGyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. Duplicate bug report detection: How far are we? *ACM Transactions on Software Engineering and Methodology*, 32(4):1–32, 2023.

[247] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust log-based anomaly detection on unstable log data. In *ESEC/FSE*, pages 807–817, 2019.

[248] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. GUIGAN: Learning to generate GUI designs using generative adversarial networks. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021.

[249] Wanlei Zhao and Chong-Wah Ngo. Scale-rotation invariant pattern entropy for keypoint-based near-duplicate detection. *IEEE Transactions on Image Processing*, 18:412–423, 2009.

[250] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232, 2019.

[251] Yinhe Zheng, Guanyi Chen, and Minlie Huang. Out-of-domain detection for natural language understanding in dialog systems. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 28:1198–1209, 2020.

[252] Wenxuan Zhou, Fangyu Liu, and Muhao Chen. Contrastive out-of-distribution detection for pretrained transformers. In *EMNLP*, pages 1100–1111, 2021.

[253] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5551–5560, 2017.

[254] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *NeurIPS*, 32, 2019.

[255] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.

VITA

Yanfu Yan

Yanfu Yan is a Ph.D. candidate in Computer Science at William & Mary, where she is advised by Prof. Denys Poshyvanyk. Her research centers on designing tailored approaches to facilitate software engineering (SE) tasks through multimodal learning, which analyzes and leverages software data in various modalities and builds upon innovative adaptation and integration of techniques from diverse fields. Yanfu's work has been published in top-tier SE venues, such as the IEEE/ACM International Conference on Software Engineering (ICSE) and the ACM International Conference on the Foundations of Software Engineering (FSE). She graduated with her M.S. in computer science and technology from the University of Chinese Academy of Sciences and her B.E. in software engineering from Xiamen University.