API UTILITY ENHANCEMENT: FROM TRADITIONAL SOFTWARE TO DEEP LEARNING FRAMEWORKS

MOSHI WEI

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY OF ELECTRICAL ENGINEERING & COMPUTER SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

APRIL 2025

Abstract

The rapid advancement of modern software technologies has amplified the reliance on APIs across diverse frameworks and libraries, enabling tasks such as data processing, system integration, and application development. APIs have become essential in accelerating innovation across domains like healthcare, finance, cloud computing, and increasingly, deep learning frameworks. However, this growing dependence on APIs has also introduced significant challenges related to their usability and reliability. Developers often face the complexity of navigating APIs that require a profound understanding of underlying principles, parameter configurations, and context-specific scenarios. Misuse of APIs can lead to degraded performance, prolonged debugging efforts, and critical application failures, posing risks across both traditional software systems and emerging deep learning applications.

Existing API recommendation systems and misuse detection tools frequently fall short due to their limited contextual understanding and lack of deep semantic interpretation. These limitations are particularly pronounced in multi-API invocation scenarios, where the dynamic interplay of APIs demands sophisticated contextual analysis. Compounding these issues are ambiguities in API documentation and the misalignment between developer intent and system-generated recommendations, which often force developers to adopt trial-and-error approaches to resolve issues.

This research hypothesizes that leveraging advanced techniques such as contrastive learning and context-aware semantic analysis can significantly enhance the accuracy of API recommendations and the effectiveness of misuse detection. To this end, a novel framework is proposed, integrating contrastive learning for precise pattern recognition, natural language processing (NLP) for deep semantic understanding, and automated error prevention mechanisms. By incorporating contextual insights, the framework aims to reduce API misuse, streamline development workflows, and improve the reliability of APIs in both traditional software and deep learning frameworks.

Empirical studies validate the proposed system's effectiveness, demonstrating substantial improvements in recommendation accuracy, error detection rates, and developer productivity. The research makes key contributions, including a taxonomy of common API misuse patterns, an advanced recommendation engine, and an automated misuse correction tool. These tools are designed to integrate seamlessly into development environments, supporting developers in building robust and efficient software systems across diverse domains.

By addressing critical gaps in API usability and reliability, this research bridges theoretical advancements in software engineering with practical development challenges. Its contributions

pave the way for more intuitive, reliable, and error-resistant tools, offering transformative benefits for developers working in both traditional software ecosystems and cutting-edge deep learning frameworks.

To my family, whose love and support made this possible.

Acknowledgements

This thesis would not have come to fruition without the support and encouragement of numerous remarkable individuals, to whom I am profoundly thankful. First and foremost, I would like to express my heartfelt gratitude to my parents. No matter where I go, they are always by my side, offering unwavering support and inspiring me to pursue my passions without hesitation. Thank you for being my pillar of strength at every stage of my life. A special thanks to my former wife, Rachel. Thank you for your companionship, encouragement, and selfless support during the most challenging periods of my life and academic career. The time we shared together was a vital source of strength that sustained me to this day. Though we did not walk side by side until the end, I will always be grateful for the light you brought into my path. I am deeply indebted to my supervisor, Professor Song Wang, whose steadfast guidance, encouragement, and support over the years have been pivotal to my academic and personal growth. His contributions have gone far beyond the role of a mentor—he has become a lifelong friend. I consider myself incredibly fortunate to have had the opportunity to work under his supervision. I would also like to extend my sincere appreciation to my PhD supervisory committee members, Dr. Zhen Ming (Jack) Jiang and Dr. Hamzeh Khazaei, for their invaluable advice, constructive feedback, and continuous support throughout my research. During my doctoral studies, I had the honor of collaborating with outstanding and dedicated researchers. My special thanks go to Professors Junjie Wang, Nachiappan Nagappan, Hung Viet Pham, Jinqiu Yang, Zhen Ming (Jack) Jiang, Hamzeh Khazaei, Maleknaz Nayebi, Alvine B. Belle, Nima Shiri, Jiho Shin, and Mohammad Mahdi Mohajer for their collaboration, insights, and encouragement. Lastly, I am incredibly grateful to my friends, including Jiho Shin, Nima Shiri, and Mohammad Mahdi Mohajer. Your friendship and generosity have made my PhD journey not only memorable but also immensely rewarding. I would like to share a message for everyone considering a PhD journey. Life is a great adventure, and in the end, it's the experience that truly matters. We can only decide whether to embark on the journey, never knowing where it will lead. So be brave, and embrace everything that unfolds along the way.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the rapid advancement of software technologies, APIs have become integral to developing and deploying complex systems across diverse domains. These interfaces, provided by widely adopted frameworks and libraries, enable developers to perform essential tasks such as **data processing**, **system integration**, and **application development**. Their accessibility and flexibility have accelerated innovation in critical fields, including **autonomous driving**, **financial forecasting**, and **healthcare**. However, this widespread adoption also introduces significant challenges, particularly in terms of **usability** and **reliability**.

The **usability** of APIs refers to how easily developers can comprehend, implement, and manage these APIs without extensive debugging or consultation of external documentation. Many APIs are inherently complex, often involving intricate configurations, multiple dependencies, and nuanced behaviors. Unlike simpler APIs that focus primarily on logic and functionality, more advanced APIs may require a deep understanding of **performance trade-offs**, **resource optimization**, and **context-specific requirements**. For developers, this complexity translates into a steep learning curve and an increased risk of errors, particularly for those new to the framework or domain.

Reliability is equally critical, ensuring that APIs behave consistently across various environments and use cases. In real-world applications, any deviation in API behavior—caused by incorrect parameter configurations, data format mismatches, or improper invocation sequences—can lead to severe consequences. For instance, in autonomous driving systems, improper API calls can result in incorrect sensor data processing, compromising the vehicle's decision-making capabilities. Similarly, in financial applications, errors in configuration can lead to inaccurate analyses, potentially resulting in substantial financial losses.

Despite the availability of **API recommendation systems** and **misuse detection tools**, their effectiveness remains limited. Current tools often rely on static pattern matching or basic rule-based systems, which lack the capability to interpret the broader context of API usage. This limitation is particularly evident in **multi-API invocation scenarios**, where APIs interact dynamically and sequentially. In such cases, existing tools frequently provide

inaccurate recommendations or fail to detect misuse altogether, forcing developers to rely on **trial-and-error** approaches. Additionally, inconsistencies in how team members select and invoke APIs can exacerbate these issues, reducing overall project maintainability.

Moreover, studies have shown that **developer intent** is often misunderstood by recommendation systems due to the ambiguity in API documentation and the lack of semantic understanding in natural language queries. For example, a developer searching for an "optimizer" specific to a particular system might receive generic recommendations that do not align with their task requirements. This misalignment between developer needs and tool outputs not only increases the likelihood of API misuse but also diminishes trust in the tools themselves.

In practice, API misuse manifests in various ways, including **parameter configuration errors**, **incompatible data handling**, and **incorrect invocation sequences**. For example, passing unsupported data types or setting inappropriate thresholds can degrade performance, cause runtime errors, or even result in critical system failures. These types of errors highlight the need for intelligent systems that can proactively identify and mitigate potential issues before they impact system reliability.

Given the growing complexity and high stakes associated with modern software development, there is an urgent need for more **advanced API recommendation systems** and **misuse detection tools**. Such tools should go beyond simple rule-based approaches by incorporating **machine learning techniques**, such as **contrastive learning**, to better understand the context of API usage and provide precise recommendations. By improving the accuracy of API recommendations and the effectiveness of misuse detection, developers can significantly reduce debugging time, improve development efficiency, and enhance system robustness.

Addressing these challenges is crucial not only for individual developers but also for organizations relying on APIs for critical systems and decision-making processes. The impact of improving API usability and reliability extends beyond code quality, influencing overall system stability, user safety, and business outcomes. Therefore, research aimed at advancing API recommendation and misuse detection systems is both timely and essential, promising to bridge the gap between the growing complexity of APIs and the practical needs of developers.

1.2 Research Hypothesis

The growing adoption of machine learning (ML) technologies across various industries has led to a surge in the use of complex ML APIs. These APIs, such as those found in TensorFlow, PyTorch, and Scikit-learn, play a crucial role in tasks ranging from data preprocessing and model training to inference and evaluation. While these APIs offer powerful capabilities, their technical depth and complexity present significant challenges for developers. Incorrect API usage can lead to suboptimal model performance, increased debugging time, and in some cases, critical system failures. Current API recommendation and misuse detection tools often fall short in addressing these challenges due to their limited contextual understanding and inability to handle complex multi-API invocation scenarios.

In traditional software development, APIs are generally more straightforward, focusing on functional implementation and logic handling. Developers primarily deal with well-defined input-output operations and can rely on comprehensive documentation or error messages to guide proper usage. However, ML APIs introduce additional layers of complexity, including hyperparameter tuning, data format requirements, and algorithmic dependencies. For instance, selecting the wrong optimizer, configuring an inappropriate learning rate, or passing incompatible data types can lead to degraded model accuracy or convergence issues. This complexity increases the likelihood of misuse, especially for developers who lack deep expertise in machine learning.

Despite advancements in API recommendation systems, most tools rely on static code analysis, pattern matching, or predefined rule sets. These approaches often fail to capture the dynamic nature of ML workflows, where API usage depends heavily on contextual factors such as data characteristics, model architecture, and task-specific requirements. Furthermore, the current systems often lack deep semantic understanding, limiting their ability to interpret developer intent expressed through natural language queries or code comments. Consequently, developers frequently encounter irrelevant or inaccurate API suggestions, leading to frustration and increased reliance on trial-and-error methods.

Hypothesis: This research hypothesizes that leveraging advanced machine learning techniques, particularly contrastive learning and context-aware semantic analysis, can significantly enhance API recommendation accuracy and misuse detection. By developing an intelligent system capable of understanding the broader context of API invocations, it is possible to reduce API misuse, improve development efficiency, and enhance the overall reliability of ML-based systems.

This hypothesis is built on several key assumptions:

1. **Contextual Understanding:** Correct API usage is highly dependent on the surrounding code context, including the sequence of API calls, data structures, and hyperparameter settings. A system capable of analyzing these elements holistically can provide more accurate recommendations and identify potential misuse with higher precision.

2. **Contrastive Learning:** Traditional rule-based systems lack flexibility and adaptability across diverse scenarios. By utilizing contrastive learning, which focuses on distinguishing between similar and dissimilar patterns, the system can effectively learn nuanced relationships within API usage, enhancing its ability to generalize across different tasks and domains.

3. **Semantic Matching and Natural Language Processing (NLP):** Developers often express their intent through natural language, either in the form of comments or search queries. Integrating semantic matching with NLP techniques allows the system to bridge the gap between human language and API documentation, improving the relevance and accuracy of recommendations.

4. **Error Prevention and Automated Repair:** By proactively detecting and classifying API misuses such as parameter configuration errors, incorrect invocation sequences, and data format mismatches—the system can offer targeted repair suggestions or even automate fixes. This not only reduces the manual debugging burden but also enhances system stability and performance.

The hypothesis further posits that integrating such an intelligent system into development environments, including Integrated Development Environments (IDEs) and Continuous Integration/Continuous Deployment (CI/CD) pipelines, can lead to a significant reduction in API-related errors across various stages of the development lifecycle. By addressing both the usability and reliability aspects of ML APIs, this approach aims to empower developers to build robust machine learning models more efficiently, ultimately driving innovation and reducing time-to-market in ML projects.

1.3 Thesis Overview

The rapid adoption of machine learning (ML) has led to a significant rise in the use of Application Programming Interfaces (APIs) across various domains, enabling developers to integrate complex functionalities with minimal effort. However, this convenience comes with the challenge of API misuse, which can lead to critical errors, degraded performance, and increased development costs. Despite the availability of extensive documentation and community support, developers often encounter difficulties due to the evolving nature of APIs, lack of contextual guidance, and insufficient automated tools for error detection.

This thesis aims to address these challenges by developing a machine learning-driven framework that enhances API usability through intelligent recommendation and misuse detection. Specifically, it explores the integration of contrastive learning and natural language processing (NLP) techniques to provide developers with context-aware, semantically accurate API suggestions and proactive error identification.

The thesis is structured into three key parts:

1. **Problem Definition and Contextual Analysis:** This section introduces the problem of API misuse, providing a comprehensive analysis of the factors contributing to this challenge. It examines common API misuse patterns, their implications for software development, and the limitations of existing solutions. The discussion highlights the increasing complexity of APIs across domains, emphasizing the need for an advanced system capable of interpreting code context and aligning with developer intent.

2. **Proposed Framework and Methodology:** This section details the design and implementation of the proposed framework. It introduces a novel contrastive learning approach to differentiate between correct and incorrect API usage, complemented by NLP techniques for semantic understanding of code and associated documentation. The framework's architecture, data collection process, and model training strategies are outlined to ensure precise API recommendations and proactive error detection.

3. **Evaluation and Results:** This section evaluates the framework's effectiveness through empirical studies conducted on diverse codebases. Key performance metrics, including recommendation accuracy, error detection rates, and improvements in developer productivity, are analyzed. Additionally, the integration of the framework into development environments, such as IDEs and CI/CD pipelines, is assessed to demonstrate its practical impact on the software development lifecycle.

This thesis contributes to both academia and industry by presenting a comprehensive solution to the persistent problem of API misuse. By enhancing the developer experience and reducing the likelihood of API-related errors, this research aims to facilitate more efficient, reliable, and scalable ML software development. It bridges the gap between theoretical advancements in machine learning and practical software engineering challenges, paving the way for future innovations in intelligent development tools.

1.4 Thesis Contribution

This thesis addresses the challenges of API usability and reliability by proposing novel solutions to enhance API usage efficiency and reduce errors. The primary contributions of this research are as follows:

1. **Development of an ML API Usability Framework** A comprehensive framework is introduced to systematically analyze and improve the usability of ML APIs. This framework evaluates API design, developer interactions, and common usage patterns, providing guidelines for creating more intuitive and error-resistant APIs.

2. **Novel API Recommendation System Using Contrastive Learning** A state-of-the-art API recommendation system based on contrastive learning is developed to assist developers in selecting the most appropriate API calls. By leveraging contextual code embeddings and semantic understanding, this system offers precise, context-aware API suggestions, significantly reducing the likelihood of incorrect API usage.

3. **Automated API Misuse Detection and Correction Tool** An automated tool for detecting and correcting ML API misuses is designed and implemented. This tool utilizes static code analysis and runtime monitoring to identify common API misuse patterns, such as incorrect parameter configurations, data format mismatches, and improper call sequences. It then generates actionable suggestions and can perform automatic corrections, enhancing development efficiency and system stability.

4. **Categorization and Analysis of Common ML API Misuse Patterns** A taxonomy of common ML API misuse patterns is constructed based on real-world case studies and developer feedback. This categorization provides insights into the root causes of API misuse, helping both developers and API designers mitigate errors and improve overall API design.

5. **Evaluation of the Proposed System's Effectiveness** The effectiveness of the proposed recommendation and misuse detection tools is evaluated through comprehensive experiments and user studies. Results demonstrate a significant reduction in API misuse rates, faster debugging times, and improved system performance, validating the practicality and impact of the proposed solutions.

6. **Contribution to the Broader Machine Learning Community** By open-sourcing the API recommendation and misuse detection tools, this research contributes to the broader ML and software engineering communities. These tools are designed to integrate seamlessly with popular development environments, fostering widespread adoption and continuous improvement.

These contributions collectively advance the state of API usability and reliability, providing both theoretical insights and practical tools to enhance the development of machine learning systems.

1.5 Related Publications

Previous versions of the research work discussed in this thesis have been distributed through the following publications, listed chronologically. My principal contributions to these works included collecting and processing raw data, performing experiments, analyzing outcomes, and writing the manuscripts. Together, these tasks represent more than 80% of the total effort dedicated to these studies.

- **Moshi Wei**, Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Lin Shi, Jinqiu Yang, Song Wang, Zhen Ming (Jack) Jiang. A Survey on Query-based API Recommendation. (Under Review of TOSEM)

- **Moshi Wei**, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, Song Wang. CLEAR: Contrastive Learning for API Recommendation. In Proceedings of the 44th International Conference on Software Engineering**(ICSE 2022)**. 12 pages.

- **Moshi Wei**, Yuchao Huang, Junjie Wang, Jiho Shin, Nima Shiri Harzevili, Song Wang. API Recommendation for Machine Learning Libraries: How Far Are We? In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering **(ESEC/FSE 2022)**. 12 pages.

- **Moshi Wei**, Nima Shiri Harzevili, YueKai Huang, Jinqiu Yang, Junjie Wang, Song Wang. Demystifying and Detecting Misuses of Deep Learning APIs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering***(ICSE 2024)**. 13 pages.

The following papers were published in parallel to the publications mentioned above. These studies are not directly related to the usability and reliability of software APIs, however, they explored how to improve the deep learning software API-related infrastructure.

- Huang Yuekai, Wang Junjie, Wang Song, **Wei Moshi**, Shi Lin, Liu Zhe, and Wang Qing. Deep API Sequence Generation via Golden Solution Samples and API Seeds. ACM Transactions on Software Engineering and Methodology **(TOSEM 2024)**. 20 pages.

- Jiho Shin, Hadi Hemmati, Moshi Wei, and Song Wang.Assessing Evaluation Metrics for Neural Test Oracle Generation. IEEE Transaction on Software Engineering **(TSE 2024)**. 11 pages.

- Nima Shiri Harzevili, Mohammad Mahdi Mohajer, Moshi Wei, Hung Viet Pham, and Song Wang. History-Driven Fuzzing For Deep Learning Libraries. ACM Transactions on Software Engineering and Methodology **(TOSEM 2024)**. 28 pages.

- Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, Song Wang. Effectiveness of ChatGPT for Static Analysis: How Far Are We? Proceedings of the 1st ACM International Conference on AI-Powered Software **(AIware 2024)** 10 pages.

- Jiho Shin, Moshi Wei, Junjie Wang, Lin Shi, and Song Wang. The Good, the Bad, and the Missing: Neural Code Generation for Machine Learning Tasks. ACM Transactions on Software Engineering and Methodology **(TOSEM'23)**. 20 pages.

1.6 Thesis Organization

This thesis is structured into multiple chapters, each addressing specific aspects of the research problem and contributing to the overall understanding of API recommendation systems. Below is a comprehensive outline of the thesis organization.

Chapter 1: Introduction. The first chapter introduces the research problem, providing the context and motivation behind this study by discussing the importance of API recommendation in modern software development and highlighting existing gaps. It outlines the research hypothesis, giving a clear direction for the study, and presents an overview of the thesis, detailing the key contributions while outlining the structure of subsequent chapters.

Chapter 2: Literature Review. This chapter provides an extensive review of literature related to API recommendation systems, covering foundational concepts such as APIs, API usage pattern mining, SDKs, ASTs, and API recommendations. It describes the methodology for collecting and analyzing research, examines trends in API recommendation, discusses challenges in datasets, reviews modeling approaches like machine learning and deep learning,

and explores evaluation metrics, comparing automatic and human evaluation methods. The chapter concludes with a summary of open problems and future research directions.

Chapter 3: Recommending APIs Using Supervised Contrastive Learning. Chapter 3 introduces a novel approach leveraging supervised contrastive learning, beginning with an overview of contrastive learning techniques, such as bag-of-words models, language embeddings, and joint embedding training. It details the model-building process, API search mechanism, and adaptations for class-level recommendations, followed by a description of the experimental design, dataset, evaluation queries, and performance measures. The chapter discusses the proposed model (CLEAR), evaluates its effectiveness against baselines, explores its real-world applications, and concludes with an analysis of threats to validity and related work.

Chapter 4: Understanding and Mitigating the Uniqueness of Machine Learning API Recommendation. This chapter identifies and addresses unique challenges in machine learning API recommendation, starting with an empirical study of research questions and Stack Overflow posts. It analyzes the performance of existing approaches on Python-based machine learning questions, introduces the FIMAX algorithm for frequent itemset mining, and evaluates its effectiveness across various settings. The chapter concludes with a discussion of generalizability, reasons for performance declines, and threats to validity.

Chapter 5: Demystifying Machine Learning API Misuse. This chapter investigates the misuse of machine learning APIs in software development, identifying common patterns of misuse, exploring their causes, and proposing mitigation strategies. By addressing these issues, the chapter aims to improve API usability and reduce errors in machine-learning applications.

Chapter 6: Conclusion and Future Work. The final chapter summarizes the key findings of the research, highlighting contributions to the field of API recommendation while outlining potential future work. It discusses areas for further exploration and improvement, including the adoption of emerging technologies to enhance API recommendation systems.

Chapter 2

Literature Review

2.1 Introduction

Choosing the appropriate software API is crucial in ensuring the successful implementation of software systems. API recommendation tools provide a service that identifies and suggests the appropriate APIs to use, based on the specific requirement and objectives of a project [1]. API methods are used by developers to connect different software programs through abstraction, serving as a glue that connects various software libraries to an automated pipeline. However, recommending an API can be a challenging task due to the vast number of APIs available in a library, each with its unique functionality [2]. There are generally two types of API recommendation tools [3]: code completion-based [4] and query-based [5]. Code completion-based API recommendation tools act as plugins in integrated development environments (IDEs), providing auto-complete suggestions for the current code snippet. On the other hand, query-based API recommendation tools are more like search engines, In this approach, users input a query regarding the feature they intend to implement, and the tool then generates a list of relevant APIs based on the provided query. Query-based API recommendation faces significant challenges in understanding the semantics of a user query and retrieving the appropriate API for the user's requirements [6]. Advanced modeling techniques like deep learning can be useful in enabling semantic parsing capability. The field of API recommendations has experienced significant development and refinement over the past 10 years [7, 8, 6, 9, 10]. New computer technologies and algorithms have had a profound impact on API recommendations. The underlying technologies for API recommendations have evolved from early API usage pattern analysis [11, 12] to later deep learning-based API recommendations [10, 7] and most recently, API recommendations based on large language models [13]. Despite significant changes in the API recommendations field, certain patterns have been identified that are helpful for future research directions and address pending technical challenges, especially through recognizing patterns from existing studies. However, existing surveys on API recommendation primarily focus on general API recommendation or code completion-based systems. There is a noticeable gap in surveys specifically addressing query-based API recommendations, particularly concerning the architecture, methodologies,

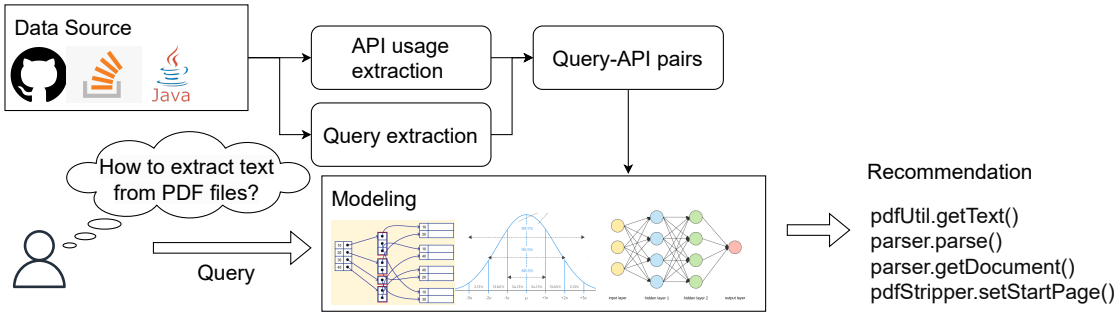

Figure 2.1: The overview of query-based API recommendation.

and applications of these tools. For example, many studies have found that relying solely on API documentation is insufficient for accurate and comprehensive API recommendations. This is because user queries often differ in style from the textual descriptions found in API documentation. One key factor in building an effective query-based recommendation system is bridging the gap between the representation of user queries and the descriptions or source code of the API [14, 15, 16, 17]. Consequently, considerable effort has been invested in collecting new data sources, with some typical sources being Stack Overflow, GitHub, and the Android app store.

As shown in Figure 2.1, a typical API recommendation approach often contains the following steps. **Data Collection:** The first step toward building an API recommendation model is to collect data. Multiple sources offer datasets for API recommendation, such as GitHub [18, 19, 20], StackOverflow [21, 22, 23], and online documents [24, 15]. Researchers often collect new data in experiments rather than reusing existing standard datasets due to the dynamic nature of software APIs, which continuously evolve through version updates. **API Usage Extraction and Query Extraction:** Once the data is collected, it needs to be processed into query-API pairs for modeling purposes. API extraction techniques fall into two groups based on the data source: **single-API usage** information and **multi-API usage** information. The latter is also named **API usage patterns**, representing sequences of frequent API method calls extracted from software source code. This process requires appropriate extraction techniques such as document parsing [25, 26, 5, 27, 28, 29, 30, 21, 17, 31], graph analysis [32, 33, 34, 35, 14, 15, 36], and AST parsers [37, 38, 39, 40]. Modeling: Modeling is the most evolved process in API recommendation. Techniques range from statistic-based approaches such as indexing [41, 30, 42, 8, 34, 24, 43] to ML approaches such as Word2Vec [44, 45, 46, 47], and the latest DL techniques such as LSTM [48, 19, 7, 49] Evaluation: Once the modeling is finished, the model's performance is evaluated using a separate test dataset. Various metrics, such as accuracy, precision, recall, and F1 score, are calculated to assess the model's effectiveness. The model may also be validated against real-world API queries to measure its practical usability [50, 30, 5]. In this survey, 34 studies were collected from 19 conferences and journals in the past ten years (2012 - 2023). This thesis investigated the existing studies on API recommendation and proposed a list of research questions (RQs) to summarize this thesis's findings. The RQs include trends in

API recommendation studies, common data sources in API recommendation tools, common data processing techniques in API recommendation tools, Common modeling techniques in API recommendation tools, and Common evaluation processes in API recommendation tools. Based on the identified gaps and technological opportunities in the literature, this survey also identifies open issues and challenges in query-based API recommendations. The following are the details of each RQ.

- **RQ1.** *What are the trends in API recommendation studies?*
  - What is the distribution of publication time in API recommendation studies?
  - What is the distribution of publication venues in API recommendation studies?
  - what are the frequent key directions and concepts in API recommendation studies?
- **RQ2.** *What are the common data sources in API recommendation tools?*
- **RQ3.** *What are the common data processing techniques applied in API recommendation tools?*
  - What are the common API usage pattern types?
  - What are the common API usage extraction methods?
- **RQ4.** *What are the common modeling techniques in API recommendation tools?*
- **RQ5.** *What are the common evaluation processes in API recommendation tools?*

This chapter makes the following contributions:

- To the best of knowledge, compared to the related surveys [3, 51, 9, 52, 53] study query-based API recommendation systems and their subsystems, including data sources, data extraction, modeling, and evaluation metrics.
- This thesis provided a classification of modeling techniques used in API recommendation based on their architectures, along with an analysis of the correlation between modeling techniques and the data extraction method employed in these models.
- This thesis analyze the challenges and opportunities inherent in the current state of API recommendation research and provide directions for future API recommendation research endeavors.
- This thesis have shared this thesis's results and analysis data as a replication package to allow other researchers to follow this study and extend it.

This survey aims to provide software engineering developers with an overview of the current status of query-based API recommendation tools. It also serves as a guideline for researchers who want to understand the landscape of query-based API recommendation studies and become aware of the challenges and opportunities in the literature on this topic.

2.2 Background

Software developers use APIs to enable communication between software components. In software research, API usage pattern mining [54] is commonly employed to analyze how developers utilize APIs. This analysis reveals patterns that contribute to improved API design and usage. To assist developers in working with APIs, modern software development practices provide a Software Development Kit (SDK), which includes tools, libraries, and documentation that simplify the integration of an API into a project [55].

When developers seek to understand APIs within a library, they often rely on Abstract Syntax Trees (AST) to examine the code's structure [56]. An AST is a hierarchical representation of the syntactic structure of code. By parsing ASTs [57], developers can identify API usage patterns, helping them optimize their development workflows. Additionally, API recommendation systems can suggest relevant APIs to developers based on historical usage patterns, community preferences, and project requirements, further streamlining the development process. The following section provides a detailed discussion of each of these concepts.

2.2.1 Application Programming Interface

In the software engineering domain, an API represents a set of methods designed for communication between software programs. Software libraries under active development usually update their API regularly, ensuring its alignment with evolving requirements and support of new features. The risk of software failure increases during this practice due to multiple factors such as version mismatch, inefficient method execution, and security vulnerabilities[58]. Version mismatch is a common problem that users encounter when the API in the latest version is incompatible with its code base [59]. This can result in software failures. In addition, the latest version of API may contain inefficient methods, which can cause dead loops or infinite execution times. Insufficient testing and design validation may also result in security issues [60]. It should be noted that maintaining the API document requires a considerable amount of effort from the library developer.

In 1991, Malamud [61] introduced the term API as "*a compilation of services accessible to programmers, intended for the execution of specific tasks.*" The phrase "*API method call*" refers to the act of invoking an API method within the source code. API method calls share similarities with traditional method invocations, but they also possess unique attributes. APIs are external methods that are exposed to end-users for software development and integration. APIs consist of two types of public methods: functional API and communication API. Functional APIs are designed for specific tasks, while communication APIs transmit data between objects to streamline communication among software applications. On the other hand, a method refers to a general function within the source code that is responsible for executing a specific action or task. A method can be either internal or external to the software application.

2.2.2 API Usage Pattern Mining

Creating a recommendation system requires first analyzing the practical usage of APIs. Collecting enough data would enable us to identify recurring usage patterns. This practice is called API usage pattern mining. API usage pattern mining involves collecting an extensive repository of source code snippets and utilizing specialized tools to identify recurring usage patterns within these code fragments [54]. The goal of API usage pattern mining is to identify recurring sequences of API method calls. For instance, when a developer wants to update a string in a file, an API usage pattern would provide recommendations that suggest the use of a file object to access the file, then create a file writer function for the string write operation, and finally call the close function of the file object after the operation to close the file. Within an API usage pattern, a user can expect at least one frequently occurring sequence of API method calls [62]. Furthermore, API usage pattern mining techniques can be applied to real-time systems as well. Such systems require rapid response times and optimal memory efficiency to effectively fulfill their operational requirements. Therefore, it is critical to analyze the performance bottlenecks of their APIs through API usage pattern mining [63].

2.2.3 Software Development Kit

A Software Development Kit (SDK) represents a more comprehensive toolkit for crafting applications on specific platforms compared to an API, which primarily facilitates system-to-system communication. In addition to APIs, SDKs frequently encompass supplementary libraries, tools, documentation, and code samples. SDK significantly diminishes the complexity and learning curve associated with the seamless integration of third-party functionalities into applications. Notably, SDK developers usually maintain a community that provides code samples and tutorials to facilitate developers in gaining proficiency in utilizing the SDK's APIs [27].

Furthermore, some SDKs offer functionalities that extend beyond the scope of basic functional and communicative APIs. For instance, the Android SDK [64], designed for Android application development, offers not only functional APIs but also delivers pre-defined UI components, debugging tools, and emulators to provide extensive support for application development, which demonstrates the depth of support SDKs can provide. SDKs commonly feature a comprehensive ecosystem where users can actively contribute by building and sharing customized modules. This community-driven aspect not only benefits the versatility of the SDK itself but also establishes a robust sense of community around its usage. This collaborative ecosystem becomes a valuable asset that nurtures innovation, sharing of insights, and collaborative problem-solving among developers.

2.2.4 Abstract Syntax Tree

The Abstract Syntax Tree (AST) functions as a tree representation of a code snippet's source code, effectively conveying the structural information of the source code. Each node within this tree representation corresponds to an entity present in the source code. Abstract Syntax

Trees (ASTs) play a crucial role in software API recommendation systems by providing a structured, hierarchical representation of code. ASTs enable these systems to analyze the code's context and syntax, allowing for more accurate and context-aware API suggestions. By traversing the AST, the system can understand code patterns, such as loops or data processing, and recommend relevant APIs accordingly. Additionally, machine learning models can leverage ASTs to learn from past API usage and make informed recommendations. AST can be extracted by parsing the source code using an AST parser. The AST parser is a tool that reads through the source code and converts it into a tree structure based on syntax rules. However, there's no guarantee that parsing will be successful. To ensure successful parsing, the code must be complete and compilable. Also, these ASTs are often used as input in expressing a program's data structure for an API recommendation system. For example, in source code refactoring, developers can use the structural information in the ASTs to perform effortless transformations and refactor the code snippet. By examining the AST of a program without executing the code, automated code analysis tools can effectively detect syntax-related issues. In API recommendation systems, ASTs frequently serve as a foundational data representation, representing structural information from the source code [24, 65, 48, 19].

2.2.5 API Recommendation

API recommendation tools provide users with suggestions for suitable API methods based on their input. Broadly, these tools can be categorized into two main types. The first type is query-based API recommendation, which operates similarly to intelligent search engines. These systems suggest specific API method names in response to programming queries submitted by users. For instance, *DeepAPI* [7], proposed by Gu et al., recommends an API method by learning from extensive API sequence data using an LSTM model. When a user submits a programming query such as "*How can I sort an ArrayList in Java?*", *DeepAPI* invokes its deep learning model to transform the text sequence into a text embedding tensor and subsequently returns the suggested API method `ArrayList.sort` based on either a sequence generation or text classification model. This type of recommendation tool relies on the textual content of the query to deliver relevant suggestions.

The second category of API recommendation tools involves code completion utilities [4], designed to facilitate developers by offering recommendations for the next API method based on the context provided by the current API. An illustrative example is the code completion tool *PyReco* [66], which incorporates API recommendation capabilities. *PyReco* operates in real-time, suggesting API methods that are contextually relevant to the code being written. For instance, when the source code contains statements like `file = "samples/sample_file.txt"` and `f = open(file, "w")`, *PyReco* intelligently suggests `f.readlines()` as the next API method to consider. Typically, these code completion tools seamlessly integrate into popular development environments such as IntelliJ IDEA or Visual Studio Code, enhancing the coding experience by offering timely and context-aware API recommendations.

2.2.6 System Overview

illustrates the workflow of the query-based API recommendation tool, comprising six key components: **Data Source**, **API Usage Extraction**, **Query Extraction**, **Modeling**, and **Evaluation**.

Data Collection: Query-based API recommendation tools usually collect data from open source code repositories such as GitHub or online source code forums such as Stack Overflow [8]. Such data sources are ideal for collecting query text and corresponding source code because they contain a substantial amount of API usage-related information [67]. Researchers leverage multiple sources for building datasets for API recommendation, including Stack Overflow [8], GitHub [7, 49], and Official API Documents [14], to gather data for building the dataset. Most recommendation tool creates their specialized dataset tailored to the specific programming language or domain it serves. For instance, a Java API recommendation tool's knowledge base primarily contains information about Java API usage.

API Usage Extraction and Query Extraction: Before modeling, the API data collected from the data source requires two types of pre-processing in parallel, which are API usage extraction and query extraction. The choice of API usage extraction techniques depends on the modeling input [68]. For instance, a graph-based API recommendation tool utilizes graph-based modeling and thus requires graph input. In such cases, the selected API usage extraction technique transforms the data in the knowledge base into a graph format. Conversely, many API recommendation tools adopt code processing techniques focused on extracting API method names from code snippets or documents [69, 70, 62]. Common processing methods include Graph Analysis [34, 71, 72, 73], and Abstract Syntax Tree (AST) parsing [74, 75, 19, 76]. The modeling component of the query-based API recommendation tool requires both query data and API data in pairs. The queries are collected from the same data source from which the code snippet is collected. If the data source is StackOverflow, then the query will be the title of the post. If the data source is from GitHub, then the query is synthesized from the comments or method name of the method. Similar to API usage extraction, the query collected may not be directly used in training, this is because users often have multiple ways of expression for the same question, and statistical API recommendation tools such as RACK [8] may not support such flexibility in querying a recommendation system.

Modeling: Modeling is another core component of each API recommendation tool. It is the technique used to convert instances or queries in a knowledge base into abstract representations for computation and comparison. Existing research in the field of API recommendation predominantly employs modeling techniques including statistics machine learning and deep learning. The techniques for API recommendation models include TF-IDF vectors, Graph Neural Networks (GNNs) [14], Collaborative Filtering (CF) [77], Recurrent Neural Networks (RNNs) [78], and Transformers [6]. A detailed discussion of Modeling can be found in Section 2.7.2.

Evaluation: API recommendation tools usually employ evaluation metrics derived from information retrieval techniques. Since the majority of programming queries can be adequately addressed with just one or two APIs [79], most API recommendation tools prioritize the

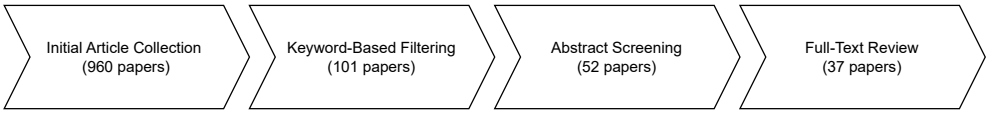

Figure 2.2: The selection process of collected research articles

correctness of the first or the initial few answers. Therefore, it is logical to evaluate tool performance using information retrieval metrics [80]. For API recommendation tools that propose sequences of APIs, BLEU serves as a measure to assess the quality of the generated sequence relative to the desired answer [7].

2.3 Overview

2.3.1 Survey Process

To conduct the survey, this thesis followed the systematic survey guidelines of Kitchenham et al. [81] to collect relevant papers published from 2012 to 2024. To begin with, a set of search terms was formulated using 10 API recommendation paper titles familiar to the research group.

Figure 2.2 illustrates the paper selection process. First, relevant papers were collected using positive search terms across multiple bibliographic databases, including DBLP, arXiv, Google Scholar, and IEEE Xplore, yielding an initial set of 960 papers. To address the limitations of these search engines, a strict keyword-matching process was applied to the paper titles, reducing the dataset to 101 papers. Following this, the abstracts were manually reviewed to determine their relevance to query-based API recommendation, leaving 52 papers. For papers where the abstract did not clearly indicate relevance, a full-text review was conducted, resulting in a final set of 37 papers directly related to query-based API recommendation research. The challenges and detailed steps of this filtering process are discussed in Section 2.3.3 and Section 2.3.4. Table 2.1 lists the positive and negative keywords used in this process.

Next, information was extracted from each paper using a predefined attribute list based on reading and analysis of common components from a random subsampling of papers. The challenges encountered during this process and the solutions implemented are detailed in Section 2.3.5. Through comprehensive analysis, this study addresses research questions (RQs) such as the main trends in API recommendation research, common components of existing API recommendation tools, unique features of current API recommendation tools, and typical evaluation processes in API recommendation research. The following sections describe the methodology in detail.

2.3.2 Datasource Selection Process

Figure 2.2 shows the overview of the article selection process. The positive search terms were initially identified through a manual investigation of several well-known API recommendation

Table 2.1: Key Words for Inclusion and Exclusion

Decision	Key Words
✓	API recommendation, API learning, API search, API retrieval
✓	API method recommendation, API Q & A, API question answering
✗	Program repair, API misuse, Malware, Bug fix, Web API
✗	Error message, API migration, Vulnerability detection

papers, such as BIKER [21], RACK [8], DeepAPI [7], and CLEAR [6]. Subsequently, online searches were conducted to identify appropriate academic databases, and the selected search terms were used to gather relevant articles. The databases chosen for this study included Google Scholar, arXiv, DBLP, and IEEE Xplore as sources for paper collection.

2.3.3 Search Term Collection

The identification of search terms began with a manual review of the top ten most cited results from Google Scholar for the query *API recommendation*. Each paper's relevance to API recommendation was carefully evaluated. From these papers, ten keywords were manually extracted for use as search terms. These keywords were subsequently applied across multiple academic databases, including IEEE Xplore, arXiv, DBLP, and Google Scholar. The search collected papers containing the selected keywords in their titles or abstracts, using the logical operator `OR` (e.g., *API recommendation* OR *API method recommendation*). This initial search yielded 960 papers. Manual investigation revealed a significant number of irrelevant papers due to individual token matches. To improve precision, the search strategy avoided matching individual words separately, as generic terms like *API*, *method*, and *recommendation* produced many off-topic results when searched in isolation. Since search engines typically match both complete terms and individual tokens by default, additional refinement became necessary. Further analysis involved manual examination of paper abstracts to compile comprehensive inclusion and exclusion criteria. A strict keyword-matching process was then implemented, filtering paper titles based on exact positive keywords while excluding those containing specified negative keywords. Table 2.1 presents the complete list of keywords used for inclusion and exclusion during the paper selection process. After applying this keyword-matching filter, 101 papers remained.

2.3.4 Paper Selection Criteria

After the initial collection of papers, the next step involved establishing a set of filtering criteria. The review team, consisting of two Ph.D. students and experienced developers, collaborated to define these criteria. For instance, selected papers were required to be either conference or journal publications proposing new tools. A comprehensive list of inclusion and exclusion criteria was developed, as presented in Table 2.2.

Table 2.2: Criteria of Inclusion and Exclusion

Decision	Criteria
✓	Papers that are published in top-ranked conferences or journals.
✓	The paper must propose a new tool with a complete evaluation and discussion on its performance and impact.
✓	The paper must be published after 2012
✓	The paper must specifically focus on query-based API recommendation
✗	Papers that are not proposing new techniques or do not have enough evaluation, including review papers, workshop papers, short papers, keynotes, and posters.
✗	The paper is an extended paper (keep only the initial version).

Following the application of these criteria filters, all papers underwent another round of manual examination. During this phase, each paper's abstract was carefully reviewed to verify its relevance to query-based API recommendation research, resulting in an initial selection of 52 papers. For cases where the abstract alone proved insufficient for decision-making, a thorough review of the full content was conducted, yielding a final selection of 37 papers.

2.3.5 Data Extraction and Analysis

The remaining 37 papers were carefully reviewed to extract data relevant to the research questions. Analysis revealed that API recommendation tools generally follow a pipeline consisting of data collection, data preprocessing, modeling, and evaluation. This framework naturally guided the design of the research questions, with an additional question included to explore trends in API recommendation. The papers were distributed among expert reviewers, with each reviewer assigned five papers for detailed analysis. Following individual analysis, a team discussion was conducted to identify common themes across the papers, which contributed to refining the research questions.

To facilitate data collection, a structured form was created based on the research questions. The review panel, consisting of two Ph.D. students, was responsible for completing this form while reviewing each paper. The research focused on four key areas: publication trends, data collection processes, modeling techniques, and evaluation methods. This information was captured in a spreadsheet, with rows representing the papers and columns indicating the specific data to be collected. During the review process, additional information valuable for the analysis was identified. These columns were subsequently added to the spreadsheet, and the previously reviewed papers were revisited to update the missing data.

2.3.6 Study Quality Assessment

To better assure the quality of the approach, a bias assessment was conducted. Specifically, a systematic approach was adopted for completing the assessment form. Initially, three authors worked independently to extract relevant information from each paper and categorize the papers based on the major category of each component of the API recommendation

system. These categorizations were then manually verified for accuracy. Additional authors participated in manually reviewing the forms completed by the first three authors, resolving any disagreements that occurred during the process to ensure consensus among reviewers.

2.4 Summary

This chapter presents a systematic survey on query-based API recommendation systems, focusing on research conducted between 2012 and 2024. Following the guidelines of Kitchenham et al. [81], the study employed a multi-phase methodology involving search term formulation, datasource selection, paper filtering, data extraction, and quality assessment.

The survey began with the identification of search terms from 10 prominent API recommendation papers. Using these terms, a comprehensive search was conducted across major academic databases, including DBLP, arXiv, Google Scholar, and IEEE Xplore, resulting in an initial collection of 960 papers. To refine the dataset, a strict keyword-matching process was applied to paper titles, reducing the number to 101. Manual reviews of abstracts further narrowed this to 52 papers, and subsequent full-text reviews yielded a final set of 37 papers directly related to query-based API recommendation.

The search terms were carefully crafted to avoid irrelevant matches from generic keywords like *API* or *recommendation*. This was achieved through the application of exact keyword matching, supplemented by inclusion and exclusion criteria. The detailed filtering process, supported by Tables 2.1 and 2.2, ensured the relevance of selected papers.

A structured approach to paper selection involved collaboration with a review team comprising Ph.D. students and experienced developers. Each paper was assessed based on strict inclusion criteria, such as being a conference or journal paper and proposing novel tools. Data extraction focused on four key areas: publication trends, data collection methods, modeling techniques, and evaluation processes. The extracted data guided the formulation of research questions aimed at identifying trends, common components, unique features, and evaluation practices in API recommendation research.

To ensure the accuracy and reliability of findings, a bias assessment was conducted. Three authors independently categorized and verified data from each paper, with disagreements resolved through consensus. This rigorous process strengthened the credibility of the survey, providing a comprehensive overview of the state of query-based API recommendation systems.

2.4.1 Comparing Against Existing Surveys

Table 2.3 presents existing surveys related to API recommendations or closely related topics. A series of review papers [3, 51, 9, 52, 53, 82] were collected and compared to highlight their key differences. The rows represent the papers being compared, while the columns denote comparison features, including data sources, evaluation methods, models, and representations. The final column categorizes each paper as either a code search paper or an API recommendation paper.

Table 2.3: Comparison with Existing Surveys

Paper	Data source	Evaluation	Model	Representation	Subject
Xie et al.[51]	✓	✓	✓	✓	Code Search
Liu et al.[53]		✓	✓		Code Search
Rahman et al.[82]		✓		✓	Code Search
Allamanis et al.[9]			✓	✓	Code Models
Shin et al.[52]				✓	Code Models
Peng et al.[3]	✓	✓			API Recommendation
Ours	✓	✓	✓	✓	API Recommendation

Among these studies, only one study is found directly related to API recommendation. Peng et al. [3] reviewed API recommendation challenges, noting that the increasing number of APIs has made it difficult for developers to find appropriate options. Although researchers have worked to improve API recommendation methods, the absence of a uniform definition and standardized benchmarks has complicated model evaluation. Their review focuses primarily on benchmarking existing tools, leaving areas like data source collection and input/output forms unexplored. In contrast, this study takes a broader approach to API recommendation, examining models and representations that address questions left unanswered by previous work, including a deeper exploration of the reasoning behind data collection and query-based recommendation.

A common method for improving code search involves enhancing natural language queries with API methods or class names extracted from platforms like Stack Overflow or GitHub. Researchers have developed algorithms that automatically reformulate queries by identifying relevant API classes and expanding the query with semantically related API methods or class names. Xie et.al. [51] surveyed deep learning-based code search including several API recommendation-based approaches code search tools. Such tools recommend APIs based on user input and then use API as the index for further code snippet retrieval. This survey focuses only on an indexing-based approach and misses other important approaches such as end-to-end generative approaches. Allamanis et.al. [9] conducted a survey of recent research at the intersection of machine learning, programming languages, and software engineering, which has proposed learnable probabilistic models of source code that leverages API usage patterns in code. The survey contrasts programming languages with natural languages and discusses how their similarities and differences drive the design of probabilistic models. The authors present a taxonomy of the models based on their design principles and use it to review the literature, as well as discuss challenges and opportunities in applying these models to various application areas. This review explores the adaptation of these models to different applications, addressing challenges and opportunities in the process, while this research focuses on API recommendation methods and applications. The existing literature review primarily focuses on the end-to-end generation of source code from natural language inputs. For example, Shin et.al. [52] surveyed source code generation models for natural language inputs, and analyzed the current trend, suggesting future research directions, including

customizing language models and exploring better source code representations. This survey also discusses embedding techniques and pre-trained models for Code models. Such techniques have been successful in natural language processing tasks. Compared to this thesis's research, this review focuses more on the code generation task, while this thesis focuses on API method recommendation. In the aspect of code search, Liu et al. [53] conducted a systematic review of 81 studies on code search to understand research trends, analyze key components of code search tools, and classify existing tools based on their focus on supporting different search tasks. Their findings highlighted outstanding challenges in existing studies and provided a research roadmap for future code search research. Rahman et al. [82] conducted a systematic review in 2023 focusing on automated query reformulation in source code search. Their study analyzed the query reformulation techniques employed by various code search tools, evaluating 70 primary studies that they categorized into eight major methodologies using a qualitative approach. They examined query reformulations targeting different types of code search, including bug localization, concept location, and Internet-scale code search. Their findings emphasize the importance of query reformulation techniques for API recommendations, particularly within the context of Internet-scale code search. In addition to detailing these methodologies, the researchers identified several limitations in existing research and proposed future opportunities for enhancing query reformulation techniques in this domain.

The difference between this thesis to the previous study is that this thesis concentrates on the standard method for developing query-based API recommendation tools. This thesis has analyzed and classified each of the key aspects involved in building such tools, including data source, evaluation, molding, and representation. Other studies, on the other hand, tend to focus more on general API recommendation tools or other related topics like code search and code modeling.

2.5 API recommendation Trend

This thesis extracts information related to publication time and venue in the 37 API recommendation papers and analyzes the trends in API recommendation studies. This examination focused on the following two key dimensions.

2.5.1 Publication Time distribution

Figure 2.3 shows the annual quantity of API recommendation studies. It demonstrates an increasing interest in API recommendation over the past decade. Notably, 2018 marked the first peak in the number of publications, coinciding with the release of RACK [8] and BIKER [21], two significant contributions to the field of API recommendation. Although there was a decrease in 2019, the overall trend from 2012 to 2024 indicates an increasing level of interest in this research area. The combined number of publications in 2018, 2021, and 2024 represents an additional 48.6% of the total publications. In 2022 and 2023, there was a notable decrease in publications on API recommendations. This thesis attributes this decline to the increasing prominence of source code models and large language models, which have

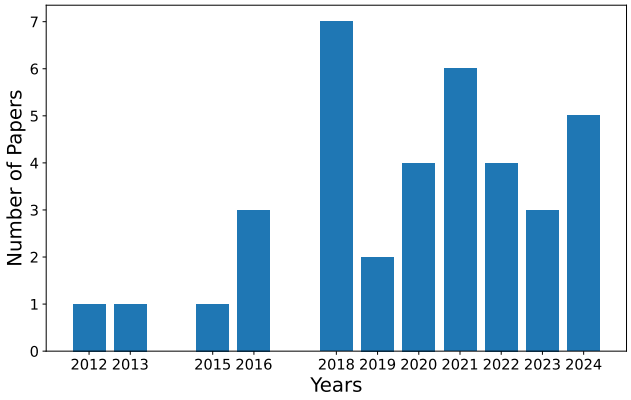

Figure 2.3: The number of papers published each year

Table 2.4: Conference and Journal names Sorted by Count

No	Acronym	Count	Full name
1	ASE	5	International Conference on Automated Software Engineering
2	ECSE/FSE	4	ACM SIGSOFT Symposium on the Foundation of Software Engineering
3	SANER	3	International Conference on Software Analysis, Evolution and Re-engineering
4	ICSE	2	International Conference on Software Engineering
5	QRS-C	2	International Conference on Software Quality, Reliability and Security Companion
6	IST	2	Journal of Information and Software Technology
7	TSE	2	IEEE Transactions on Software Engineering
8	IEEE Access	1	IEEE Access
9	APSEC	1	Asia Pacific Software Engineering Conference
10	CSMR	1	European Conference on Software Maintenance and Reengineering
11	EMNLP	1	Empirical Methods in Natural Language Processing
12	IET	1	The Institution of Engineering and Technology
13	SCP	1	Journal of Science of Computer Programming
14	TOSEM	1	ACM Transactions on Software Engineering and Methodology
15	JCST	1	Journal of Computer Science and Technology
16	MSR	1	International Conference on Mining Software Repositories
17	JSS	1	Journal of Systems and Software
18	COMPSAC	1	Computers, Software, and Applications Conference
19	IJCINI	1	International Journal of Cognitive Informatics and Natural Intelligence
20	ICSME	1	International Conference on Software Maintenance and Evolution
21	Internetware	1	Asia-Pacific Symposium on Internetware
22	ICPC	1	International Conference on Program Comprehension
23	SIGPLAN	1	Special Interest Group on Programming Languages
24	ICSME	1	International Conference on Software Maintenance and Evolution

led to improvements in complete code generation tasks. Consequently, API recommendation, as an intermediate sub-task, is not as popular as it once was. With the advent of Generative AI, API recommendation is still a crucial task. ChatGPT, for instance, faces issues with outdated knowledge [83, 84] and out-of-distribution data [85], making it less effective in code generation for new languages and libraries. Despite this, API as an index continues to play an important role in code search-related tasks. Hence, API recommendation remains a significant research area.

The publications are categorized by conference and journal names in Table 2.4. Among the 20 publication venues, the top three venues with the highest number of publications for API recommendation papers are ASE, FSE, and SANER, all of which are A+ conferences in

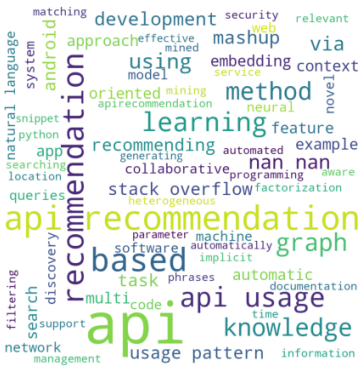

Figure 2.4: The word cloud of collected papers

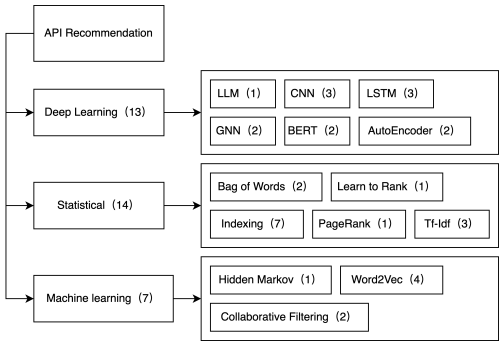

Figure 2.5: Distribution of API recommendation tools

the CORE ranking system [86]. The CORE Ranking system is widely utilized and widely recognized across various research fields, including software engineering. The number of API recommendation papers published in the top three venues comprises 32.4% of the total number of API recommendation publications.

To explore publication trends and the prevalent concepts within the API recommendation research domain, this thesis performed a keyword frequency analysis, presenting the results through a word cloud visualization. Figure 2.4 displays the word cloud chart generated from keywords in the titles of relevant papers. The word cloud provides an intuitive depiction of the main trends and key terms in this research field. Titles were tokenized, and the word cloud was created using the Python `wordcloud` package. As shown in Figure 2.4, prominent terms in API recommendation include "API usage", "knowledge", and "learning", which reflect central themes in the field. Moreover, the word cloud highlights essential components for building API recommendation tools, such as using StackOverflow as a dataset and employing natural language processing (NLP) as a modeling approach. A deeper understanding of the current direction of API recommendation research can be obtained by carefully examining all collected articles and categorizing the papers. Since all the papers are closely related to query-based API recommendations, the input format, output format, and application directions are nearly identical, making categorization based on these factors less meaningful. By comparing multiple factors, the most meaningful way to categorize the papers is based on the key differences in the modeling algorithms applied. Figure 2.5 illustrates the categorization of all papers.

Researchers generally consider API recommendation as a retrieval task rather than a generation task. In general, there are three primary directions in the modeling techniques of these tools. In the early years of API recommendation, researchers approached the API recommendation problem using statistical approaches such as Indexing [41, 30, 42, 8, 34, 24, 43], TF-IDF [15, 31, 22, 57], bag-of-words (BOW) [25], PageRank [87], and Learn-to-rank [5]. Starting in 2016, the improvements in machine learning techniques drew researcher's attention. Researchers have started applying the latest machine learning techniques such as word2Vec [44, 45, 46, 47], Collaborative filtering [50, 77], and Hidden Markov Model [20]

to API recommendation tasks. Compared to manually designed scoring functions used in ranking and evaluation, machine learning techniques like word2Vec offer more flexibility and improve the overall performance of API recommendation ranking.

In recent years, researchers have started applying deep-learning techniques such as BERT [27, 6, 88], LSTM [48, 19, 7, 49], GNN [14, 72], and CNN [89, 90, 76]. Deep learning techniques enabled more possibility for API recommendation tasks as they allow advanced semantic parsing that clusters the intention of query based on query semantics without explicit design. Also, generative models such as Large Language Models (LLM) [13] allow users to query APIs in the form of open question-answering by only giving API documents as a knowledge base context in the prompt.

2.5.2 Summary and Open Problems

This study provides a comprehensive analysis of trends in query-based API recommendation research from 2012 to 2024, based on 37 selected papers. Over the past decade, interest in API recommendation has grown significantly, with a notable peak in 2018 due to impactful contributions like RACK [8] and BIKER [21]. Although there has been a slight decline in publications in recent years, the research remains vital, especially with the rise of Generative AI models such as ChatGPT. These models face challenges with outdated knowledge and out-of-distribution data, reinforcing the importance of API recommendation as a crucial sub-task in code search and software development.

The most popular venues for API recommendation publications are ASE, FSE, and SANER, accounting for 32.4% of the total papers, reflecting the high academic quality and interest in the topic. A keyword frequency analysis highlighted central themes such as "API usage", "knowledge", and "learning", indicating a strong emphasis on applying machine learning and natural language processing (NLP) techniques to enhance API recommendation.

Despite significant advancements, several open problems remain. There is a need to improve the integration of Large Language Models (LLMs) into API recommendation systems, particularly in handling evolving APIs and real-time updates. Addressing the challenge of knowledge obsolescence through continuous learning from new API releases is essential. Additionally, hybrid approaches that balance retrieval and generation may offer more robust solutions for complex queries. The field would benefit from standardized evaluation frameworks that consider accuracy, relevance, and real-world applicability, as well as scalable solutions to efficiently handle large datasets and minimize computational overhead. Addressing these issues will be crucial for advancing API recommendation research and meeting the evolving needs of software developers.

2.6 API recommendation Dataset

The selection of an appropriate dataset represents a crucial step in developing an API recommendation system, serving as the foundational phase for creating a tool capable of recommending APIs. The choice of data source significantly influences the scope, potential

Table 2.5: Common Data Sources

Data source	Code Context	Code Completeness	Query	Quality	Quantity	Count
Github	✓	High	✗	Medium	High	12
Stack Overflow	✗	Low	✓	Low	Medium	13
Online Document	✗	High	✗	High	Low	15
Google Play,SourceForge	✓	High	✗	Medium	Medium	2

use cases, and applicability of the API recommendation. Datasets can be sourced from various origins depending on the intended application areas of the API recommendation. Table 2.5 presents the data sources utilized in the 37 collected articles. Notably, some papers incorporate multiple data sources, each of which was counted separately. Among these, GitHub, Online Documentation, and StackOverflow emerge as the three largest data sources, collectively accounting for 95.2% of the total dataset references. An analysis of these three primary data sources was conducted across five key dimensions: Code context, Code completeness, Presence of data queries, Quality of data, and Quantity of data. Each data source exhibits distinct strengths and weaknesses in these dimensions.

2.6.1 Overview of Existing Data Source

GitHub is one of the largest open-source data repositories globally, hosting over 128 million repositories, including approximately 258,000 Python and 240,000 Java repositories. Researchers typically prefer collecting data from a wide range of repositories rather than focusing on just a few. This is because GitHub encompasses a diverse array of application domains, and existing works seek to develop general API recommendation tools that can be applied across various projects, rather than being limited to project-specific recommendations.

Notably, researchers often collect GitHub datasets themselves instead of reusing existing ones [91], primarily due to the time-sensitive nature of API data. GitHub projects are frequently updated, ranging from daily to monthly changes, and the underlying code evolves over time. As a result, data collected at different points in time can yield substantially different datasets. Furthermore, some repositories may become deprecated or inactive, leading to their disappearance. To ensure that recommendation tools are built on the most current data, most studies gather up-to-date datasets.

Although individual datasets may differ, many studies use similar collection methodologies as previously published research. Compared to other data sources, GitHub offers the advantage of providing complete code context, making it particularly useful for code analysis.

GitHub is widely used for API recommendation due to its rich code context and large repositories. However, it's not universally accepted in the literature. A key concern is the contextual noise in GitHub code, which may include irrelevant information, making it harder to match APIs to specific query contexts. Stack Overflow, for example, prefers minimal working examples to avoid this issue. Another key challenge with this dataset is the absence of coding queries. The queries associated with GitHub repositories must be generated or inferred from code comments, as GitHub does not naturally pair APIs with user queries. This

introduces difficulties in forming query-API pairs [7, 49]. While GitHub provides updated repositories, online forums like Stack Overflow may offer more timely discussions on API issues, bugs, and vulnerabilities. Combining GitHub's code context with forum data could improve the accuracy and relevance of API recommendations by addressing both code and real-time usage scenarios.

StackOverflow is a popular question-and-answer forum for software developers. The data on StackOverflow is presented as posts, which consist of a question title, question description, an accepted answer, and additional answers. Researchers typically structure the data obtained from StackOverflow as question-answer pairs and use natural language processing techniques to extract features from the questions. Researchers then use mathematical modeling to establish the relationship between the questions and their corresponding code. All the datasets obtained from StackOverflow consist of question-answer pairs where the question component includes an English sentence describing an API-related question. The answer component usually contains a code snippet, which either resolves the problem or includes an API method that could potentially resolve the issue. However, it's important to note that the quality of the data varies greatly since the level of experience of each contributor varies from beginner to expert. In summary, StackOverflow fulfills the missing information of user queries, as the query can be directly collected from the title of StackOverflow posts. However, since StackOverflow is a forum for open discussion, the quality of answers varies. The answers usually contain discussions and conversations, and the completeness of code snippets is not guaranteed. As a result, filtering and post-processing tasks must be applied to StackOverflow data to improve its quality [21, 6, 8]. When comparing data from GitHub and StackOverflow, it becomes evident that StackOverflow data is more time-sensitive. This is because GitHub code is continually maintained and reviewed by developers, making it less likely to become outdated. However, as an open QA forum, StackOverflow posts are not actively updated by the community, making them more likely to become outdated [92].

In addition to sources like Stack Overflow and GitHub, online documents serve as valuable datasets. This term encompasses resources such as API documentation and online tutorials, which researchers often collect to complement other datasets. Among these, official API documentation provided by the project maintainers is considered the most reliable source of API features and descriptions, as they are manually authored and reviewed by developers. However, maintaining documentation is time-consuming and prone to becoming outdated, resulting in a relatively smaller sample size from this data source. Examples of these online documents include official API documentation, GitHub Issues, Jira tickets, and tutorials. Since these resources are manually reviewed and smaller in volume compared to data scraped from GitHub or Stack Overflow, they are typically regarded as high-quality and sometimes used as references for model evaluation [6, 14, 10, 9].

2.6.2 Overview of Existing Dataset

Figure 2.6 illustrates the scale of data collection over the years, measured by the number of unique APIs. The data reveals that the majority of studies collect fewer than 40,000 unique

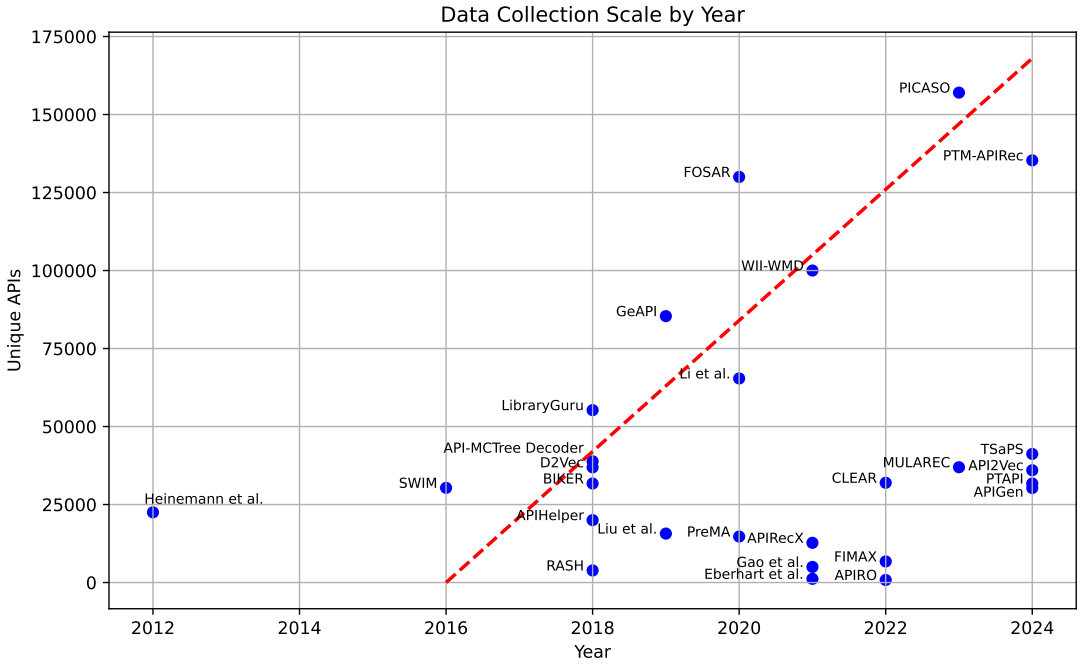

Figure 2.6: Data Collection Scale by Year

APIs. However, the read line shows a noticeable trend of increasing API volumes over time, reflecting the growing influence of big data and large-scale models on API recommendation tool development. It is important to note that the number of unique APIs was directly obtained from the original papers when available. For studies where this information was not explicitly provided, this thesis estimated the number of unique APIs as 10% of the total training dataset. Papers where this estimation could not be applied were excluded from the analysis.

Over the last 10 years, GitHub and Stack Overflow have emerged as the two largest data sources for API recommendation studies, collectively accounting for 76.4% of the total number of papers. Additionally, online documents collected from various sources are also one of the common choices. All three types of data sources have some degree of time-sensitivity. Forums are more prone to containing outdated information due to their nature. On the other hand, GitHub and online documents are less likely to have outdated data as they are regularly maintained and updated by developers.

2.6.3 Challenges: Lack of Benchmarking

The lack of common benchmarks is a significant challenge in API recommendation research. While a few studies reuse existing datasets for performance comparisons, most studies create their own. Table 2.6 shows the distribution of datasets used in existing studies. It highlights

Table 2.6: Dataset Used in Existing Studies

Dataset	Count
Self Collected	25
BIKER	7
APIBench	2
GitHub Java Corpus	2
RACK	2

that 25 studies collect their own datasets, forming the majority. BIKER, used in 7 studies, is the most frequently adopted dataset in API recommendation research. The primary reason for this reliance on custom datasets is that different tasks require various types of information, and each tool is designed for a specific purpose. Furthermore, software development is characterized by constant versioning, which complicates the reuse of existing datasets. Researchers, therefore, tend to prioritize data relevance by collecting the most recent versions of data sources using previously established procedures.

Constructing a unified benchmark dataset for API recommendation is a formidable task due to the diverse requirements of different tasks and the rapid evolution of software. The variety of programming languages and the versioning inherent in software development further complicate the creation of a single benchmark that can encompass all use cases and data attributes. However, in recent years, large language models (LLMs) have demonstrated impressive capabilities in text processing and extraction. Tools like FireCrawl[93], which convert webpages to markdown formats for LLMs to process, highlight the potential of LLMs in handling massive amounts of webpage data with flexibility across tasks. A more practical solution, therefore, is to maintain an open-source dataset, such as LiveBench[94], which includes automated web scraping pipelines to construct an evolving benchmark.

2.6.4 Challenge: Limited Dataset Quality

Ensuring high-quality datasets is a critical challenge in API recommendation research, encompassing three key sub-problems: user query quality, code context, and automated data cleaning. The first challenge lies in the quality of user queries. Platforms like StackOverflow attract developers with varying expertise and proficiency in English, leading to inconsistent query quality. The rise of large language models (LLMs) offers a potential solution, enabling the rewriting of queries to ensure consistent expression or parsing their semantics into structured data formats, serving as an intermediary step between natural language and programming language. The second issue is that only collecting API call sequences is not enough for practically useful API recommendations. Additional contextual information such as version details and API compatibility are necessary to validate the accuracy of the API sequences. Although knowledge graph techniques have been applied to automated API sequence validation to address this problem [57, 72], there is still room for improvement in this direction. The third issue concerns the cleaning of datasets. Although there are automated data cleaning methods, it is difficult to achieve a high-quality API recommendation

dataset solely through automation, especially given the challenges presented in the first two sub-problems. In some cases, manual verification is necessary to filter the test set and confirm the ground truth of the data [21, 95]. This problem is particularly noticeable in the context of query-based API recommendations.

2.7 API recommendation Modeling

2.7.1 Data Processing Overview

Preprocessing of the data collected from the data source is required before modeling for API recommendation. The data is usually in the form of query-answer pairs, where the answer part of the pair is typically a single API [21] or a list of APIs [7, 49]. To extract the API answers, one can use API usage extraction techniques such as graph analysis, AST parsing, code parsing, or document parsing. The details of these parsing techniques for each dataset are shown in Figure 2.7.

The API extraction techniques can be categorized into two distinct categories based on the data source: single-API usage information and multi-API usage information. The latter is often referred to as API usage patterns, which are sequences of frequent API method calls extracted from software source code.

For single API recommendation tasks, when building the API recommendation knowledge base, researchers often gather information related to individual API usage, including API usage examples and API method descriptions. For instance, Xie et al. [30] proposed API method recommendation techniques based on the relationship between API method names and their descriptions. Since the primary focus of the study is the analysis and reformulation of natural language descriptions of API methods, it suffices to collect only the API descriptions for knowledge base construction.

For multi-API recommendation tasks or API sequence recommendation tasks, it is essential to conduct API pattern usage mining to extract and store frequently used API sequences in the knowledge base before constructing the model. As part of the API recommendation knowledge base, API usage patterns are collected during the model construction process. These patterns complement the information gathered for single-API usage. In certain studies [67, 96, 12], the API recommendation tool also provides code snippets as API usage examples. In these cases, the tool gathers mapping information between code fragments and API method names or API usage patterns.

The API usage pattern is represented by a sequence of APIs in a code snippet. Different methods can be employed to obtain this sequence, such as graph analysis, source code parsing, and AST parsing. The goal of these approaches is to extract the API calls in the correct logical order of the source code. Additionally, documentation can also be used to obtain API sequences such as method name matches. However, the completeness of such a sequence is not guaranteed since it is parsed from document descriptions or comments rather than source code.

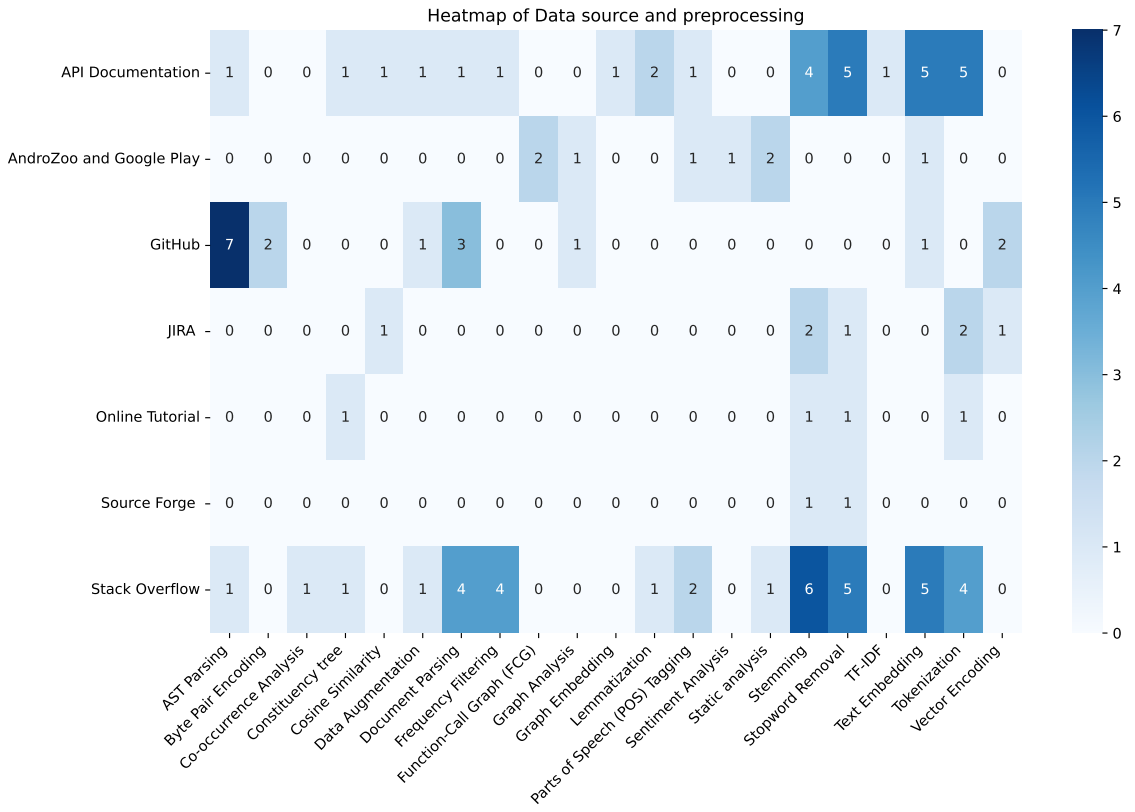

Figure 2.7: Heatmap of Data Source and Preprocessing

API usage extraction plays a crucial role in building API recommendation tools. Within the collection of papers, various model architectures and technical approaches have resulted in a range of API usage pattern collection techniques. This thesis have gathered, identified, and classified the API collection techniques used in the paper collection. To build the model, an API knowledge base is essential for recommending API methods. Researchers employ various data mining techniques to extract API usage data for constructing this knowledge base. The API knowledge base gathers API usage information from sources like API documentation and public source code archives.

Figure 2.7 presents the API usage extraction methods existing studies employ. The rows are the method employed and the columns are the data source. The graph clearly illustrates that document parsing and code parsing are the most popular tools. The following paragraph describes the data representations for API recommendation studies, including API usage extraction types, such as single API recommendation and multiple API recommendation. It also includes the types of data parsing, including document parsing and source code parsing.

Graph Analysis

Several studies have utilized graph analysis for API usage pattern extraction [34, 71, 72, 73]. Nguyen et al. [34] proposed GraPacc for context-sensitive API recommendation based on API usage patterns. They extract API usage patterns from API knowledge graphs. Nguyen et al. [71] proposed HAPI, a statistical generative model of API usages based on the Hidden Markov Model. They build the API usage representations using an API call graph model. Chen et al. [72] proposed JARST for recommending API usages by analyzing the structure information and text information of the source code. They extract API usage patterns from structured source code using graph analysis techniques.

AST Parsering

An Abstract Syntax Tree (AST) represents source code as a tree structure. To convert source code into an AST structure, the code must have a formal definition in a formal language, adhering to a set of rules defined by formal grammar [37]. AST representation is one of the most crucial static analysis techniques employed in software engineering. Multiple research fields, including code clone detection [97], defect prediction [38], vulnerability detection [39], and automated bug repair [40], have adopted the AST data representation technique.

In API recommendation, AST parsers are commonly applied when extracting APIs from Java JDK [75, 19, 76, 7, 49]. He et al. [75] proposed PyART for recommending APIs for Python programs in real-time. They extract API usage patterns by analyzing the AST nodes of Python code. Yan et al. [19] proposed APIHelper, which predicts API sequence patterns using a Long Short-Term Memory (LSTM) network. They extract API usage pattern sequences using a Java AST analyzer. Wang et al. [76] proposed plot2api, which recommends APIs based on chart plots. They extract the ground truth APIs using an AST parser. These AST parsing techniques focus on static analysis and extract API methods based on source code structure, as opposed to graph analysis techniques that monitor and extract API calls in runtime.

Code and Document Parsing

Many existing studies extract API usage information directly from API documentation [34, 71, 72, 73], primarily sourced from official documentation websites such as the Java JDK official document website, Android Developer guides, and Python library websites.

Java JDK The official Java JDK API documentation is the most frequently cited data source for API usage in this category. Its highly structured and consistent format enables automated extraction of API information through keyword matching. In existing work, parsing Java APIs from the Java JDK is typically done using Eclipse's JDT compiler. The Eclipse JDT compiler applies a static analysis approach, converting code snippets into an Abstract Syntax Tree (AST) structure, and then traversing the AST to extract the APIs. The Eclipse JDT compiler was used for document extraction. For each method, the AST

was analyzed and the JavaDoc comment extracted. Methods without JavaDoc comments are ignored. Then, the first sentence of the comment was selected as the annotation. Alternatively, API usage information can be extracted from Java libraries by parsing comments within the codebase using the JavaDoc tool. JavaDoc follows a standardized format for documenting API method descriptions. Java docstrings precede the method definition and consistently begin with `/**` and end with `*/`. Researchers can employ JavaDoc tools to extract API documentation from the codebase of any Java library that adheres to this standard.

Android Developers Guide Several studies have focused on Android SDK API methods, as documented in [18, 77, 15, 32]. Android, an operating system built on top of the Linux kernel, is primarily designed for tablets and mobile devices. The official Android SDK API reference website, known as the Android developer guide[1], provides dedicated pages for each API class, offering comprehensive details about the APIs for Researcher to extract. For example, LibraryGuru[98] extracted the core APIs, all inherited APIs, and its detailed API description from each detailed web page specifically using hypertext parsing tools.

Python Standard Library There is a growing focus on the study of Python language API method documentation, as demonstrated by prior works such as [99, 100, 101, 76, 66, 75]. Python3, released in 2008, has rapidly emerged as one of the most prominent programming languages, finding applications in diverse fields such as machine learning, deep learning, web scraping, and automation. Researchers commonly regard the Python Standard Library as the authoritative source for Python API documentation[2]. The API documentation website follows a structured format, with a dedicated page for each API class. These pages typically include details about inherited methods, usage descriptions, and occasionally, API usage examples. For instance, consider the `re.split()` function, which is a commonly used function in the Python regular expression module. On the documentation page for the re API, the `re.split()` function is enclosed within a `<code>` tag with the class name `sig-name,` indicating the function's signature name. Beneath the method name, the description of `re.split()` reads as follows: "*Split a string based on occurrences of the pattern.*" If capturing parentheses are employed in the pattern, the text of all groups in the pattern is also included in the resulting list[3]. Apart from the standard Python API documentation, there are third-party library documentation websites such as Geeks for Geeks[4]. Nevertheless, these websites often have different designs and layouts than the standard Python API library. Extracting API information from these websites requires the use of customized web scraping scripts. PyDoc is a documentation tool for Python, similar to JavaDoc. To use pyDoc, a docstring should be placed within a method's definition, and its format should start and end with triple quotation marks. If the docstring of a Python library adheres to the pyDoc standard, the pyDoc tool can be used to extract API usage information from any Python library.

[1]https://developer.android.com/reference/com/google/android/play/core/install/model/ActivityResult.html
[2]https://docs.python.org/3/library/
[3]https://docs.python.org/3/library/re.html
[4]https://www.geeksforgeeks.org/

QA Website and Online Tutorials API recommendation tools provide API methods in response to natural language queries, utilizing data collected from Online Question Answering (QA) websites [9] and online tutorials [8, 102, 32, 21]. Rahman et al. [8] establish a relationship mapping between API usage information and related programming questions by extracting knowledge from the Stack Overflow Q&A site. This approach aims to bridge the API knowledge gap that often exists between natural language questions and API usage. Sun et al. [90] create an API knowledge graph by extracting API usage patterns from Stack Overflow, enhancing the programming learning experience. Huang et al. [21] combine API documentation and programming questions from Stack Overflow to improve API recommendation ranking. Ye et al. [102] extract programming questions from Stack Overflow and utilize API documents for word similarity evaluation, specifically tailored to software engineering information retrieval tasks.

Other Tools Considering the variety of API-related Data sources, researchers apply a mixture of multiple methods for API usage extractions. There are many other tools used for API usage information extraction. For instance, He et al. [75] employ data-flow analysis for real-time API recommendation. They encode not only the API method call sequence but also information about how data is passed between objects, as the data flow may contain crucial information for the recommendation system's decision-making. Gao et al. [15] utilize APKtool to extract the resource folder and XML files from the standard Android Application distribution file (APK). From these XML files, they extract the UI element API method names and API usage patterns. Raghothaman et al. [43] utilize the standard C# AST parsing tool, Roslyn C# AST parser. The Roslyn AST parser is used in a manner comparable to the Python AST parser and the Java AST parser. Additionally, several studies employ static analysis techniques without specifying the precise tool used for API usage pattern extraction [77, 50, 103].

2.7.2 Modeling Approaches Overview

The most crucial aspect of API recommendation is the modeling process, which also plays a significant role in differentiating one paper from another in terms of novelty. This section categorizes the models used in the collected papers into several categories and subcategories, providing a detailed description of each subcategory. The three primary modeling directions are statistical approach, machine learning, and deep learning. The deep learning approach is distinguished from the other two based on whether the model contains more than three layers [104].

A significant proportion of machine learning and deep learning models are expected to be applied to API recommendation. This expectation stems from considering API recommendation research as a sub-topic of recommendation engine research, where machine learning approaches rank among the most popular and effective methods.

Table 2.7: Common Statistical Modeling Techniques

No	Model	Reference
1	Inverse Document Frequency (IDF)	[31, 105, 106, 21, 5, 107, 7, 15, 22, 108, 25, 109]
2	Indexing	[42, 8]
3	Bag of Words (BOW)	[25, 57]
4	Association Mining	[42, 110]
4	Probabilistic	[43]
5	Search Based	[111]
7	Information Entropy Influence	[112]
8	Curry-Howard Correspondence	[113]
9	Latent Dirichlet Allocation (LDA)	[46, 106]

2.7.3 Statistic-based Approach

The statistic-based models employed by API recommendation tools are listed in Table 2.7. This thesis categorizes these algorithms based on statistics into five categories. Among these approaches, the most popular model is the Indexing model. This section describes how researchers adapt each statistical model to their respective API recommendation tool.

Indexing model

The most prevalent text processing technique is indexing [41, 30, 42, 8, 34, 24, 43]. An index-based model identifies the correlation between the keywords and the APIs and combines them together to a look-up table. For a new query, the system matches the keywords in the user query to related APIs. For example, Rahman et al. [8] designed a scoring mechanism and built a keywords-API table based on the connection between keywords in the StackOverflow title and the APIs in the post. Nguyen et al. [24] constructed an indexing model to capture the fine-grained information from the code changes and recommend APIs based on the keywords in a new query.

Term Frequency-Inverse Document Frequency(TF-IDF)

The other prevalent modeling technique is TF-IDF, an algorithm for text vectorization that converts each token in a sequence into a numeric value by calculating its frequency statistics. Many researchers in the API recommendation field employ TF-IDF as a fundamental text processing and source code processing technique because TF-IDF-based techniques can be applied to general tokenizable sequences, including source code. For example, Zhang et al. [22] use the TF-IDF-based technique to transform API-related questions and functional descriptions into vectors. Gao et al. [15] utilize TF-IDF-based technique to determine the significance of APIs within their customized API set. Thung et al. [25] employ the TF-IDF-based technique to convert sequences into vectors for vector space modeling. According to this thesis, API recommendation researchers primarily use TF-IDF-based techniques for text or code vectorization and for calculating the importance score of API method calls.

Table 2.8: Common Machine Learning Modeling Technique

No	model	Reference
1	Word2Vec	[114, 115, 21, 107, 15, 5, 30]
2	DoC2Vec	[116, 46]
3	Hidden Markov Model	[71]
4	Learn to Rank (LTR)	[5]
5	Collaborative Filtering (CF)	[33]
6	Graph Embedding	[115]
7	API2Vec	[106]

Bag-of-words, PageRank, and Learn to Rank

Although most researchers approach the API recommendation task by manually designing a variety of scoring functions, there are still a few statistical approaches other than TF-IDF and Indexing. Ling et al. [57] apply a mixture of API call graph analysis and Bag-of-Words algorithm to get the graph embedding of the API sequence. Ponzanelli et al. [87] proposed an extended version of the PageRank algorithm to recommend and re-rank the content collected from online API-related sources with consideration of prominence and complementary. Zhou et al. [5] apply the Learning-to-rank (LTR) model to continuously improve the performance of API recommendation by leveraging the API-related information feature as the training data and feedback feature as the feedback of the performance improvement signal.

2.7.4 Machine Learning-based Approach

Word2Vec

Word2vec is a neural network natural language processing model proposed by Mikolov, Tomas, et al. [117] in 2013, which converts word tokens into vectors. It discovers word vectors by analyzing text sequences from a large text corpus. Initially, it was used to identify synonyms because words with similar meanings exhibit similar usage patterns in the corpus. Since similar API method calls also have comparable usage patterns, the Word2vec model can be extended to API method call recommendations in the API recommendation field. Doc2vec [118] is an extension of the Word2vec model, designed to convert documents into vectors. Huang et al. [21] proposed BIKER, a Word2vec-based API recommendation model. It models API method calls and descriptions and then recommends API method calls by comparing API method call vectors for similarity. As API recommendation research advanced, Word2vec became a baseline modeling technique for source code and language modeling [47].

Collaborative Filtering

Collaborative Filtering (CF) is a common recommendation engine technique that has been applied to a variety of recommendation tasks, including those involving movies, music, books, and online shopping. In a collaborative filtering system, there are two types of objects: items

and users. The CF system constructs a matrix based on the relationship between items and users and then ranks the vector similarity between the query item and the items in the matrix to make recommendations. There are two main types of CF techniques: Memory-based collaborative filtering and Model-based collaborative filtering.

Memory-based collaborative filtering is based on the assumption that similar users have similar preferences and, consequently, similar tastes. This approach is used to predict item ratings. Typically, the ratings in the training dataset are normalized to address the issue of rating bias. Rating bias is the observation that some users tend to give relatively low scores for all the items they review, while others tend to give high scores. To address this issue, the ratings should be normalized based on each user's rating distribution.

In API recommendation tasks, memory-based CF often encounters the sparse matrix problem due to the large number of API methods in a dataset and the fact that each code snippet typically contains only a few API method calls, leaving the rest of the vector empty. On the other hand, model-based collaborative filtering obtains item vectors through trained machine-learning models. It is more compact, faster to train, and less prone to over fitting than memory-based CF. For the API recommendation task, Wang et al. [50] used a hybrid item collaborative filtering technique. In their model, API method calls represent items, while API method declarations represent users. Memory-based collaborative filtering and model-based collaborative filtering are combined in the hybrid collaborative filtering system. Using memory-based CF, they first identify similar projects and method declarations at the project level. Then, they use model-based CF to complete the matrix. Finally, the completed API list is used to rank the candidates for API methods.

Nguyen et al. [77] proposed FOCUS, a context-aware collaborative filtering approach for extracting API usage patterns from open-source software projects and making API recommendations based on API method usage similarity. They reformulate the API recommendation problem as a collaborative filtering problem between API usage patterns as items and projects as users.

Hidden Markov Model

The Hidden Markov Model (HMM) is a state-based model trained through a Markov process. The training objective of a Markov model is to uncover a hidden state model by observing a training dataset. Although the hidden state of a Markov model is not directly observable, it can be updated using observable data for model training. Nguyen et al. [20] introduced HAPI, a Hidden Markov Model (HMM) designed to model one or multiple API method calls. Given a HAPI state, the HMM model generates a method call and then transitions in a probabilistic fashion to the next state. A custom algorithm is employed to train HAPI using API usage association information extracted from the SALAD dataset [20].

2.7.5 Deep Learning

Since the success of Deep learning-based applications such as AlphaGo [123], Convectional neural network [124] (CNN)-based image recognition, and Transformer-based language model,

Table 2.9: Common Deep Learning Modeling Technique

No	model	Reference
1	LSTM	[48, 19, 7]
2	BERT	[30, 88, 6]
3	CodeBERT	[119, 120, 121]
4	GPT	[13]
5	SBERT	[109]
6	CodeT5	[109]
7	SPT-Code	[122]
8	CNN	[89]
9	AutoEncoder	[18]
10	LLM	[13]
11	Bi-LSTM	[88]
12	Deep Q-Learning	[105]

there has been considerable interest in the field [125, 126, 127, 126, 128, 129, 130, 131, 132, 133, 134]. Deep learning has become a popular and effective technique in a variety of research fields. Deep learning has also dramatically altered the paradigm of API recommendation research.Deep learning models, such as CodeBERT and CodeT5, are often selected for building API recommendation tools due to their state-of-the-art (SOTA) performance in API recommendation tasks. These models leverage large-scale pre-training on diverse codebases, allowing them to understand code context and semantics effectively. Researchers decided to use these code-specific models because they have demonstrated superior accuracy in generating code-specific content compared to general language methods. The decision to use deep learning approaches is further reinforced by their ability to capture complex relationships within data, enabling more accurate and context-aware suggestions. This section introduces API recommendation tools that employ deep learning models, along with their implementation approaches. Among these deep learning models, LSTM, GNN, and BERT emerge as the most prevalent for API recommendation tasks.

Long Short-Term Memory (LSTM)

LSTM model is a type of recurrent deep learning model. In a recurrent model, the term "recurrent" signifies that the output of a recurrent layer is looped back to its input. The key distinction between the recurrent model and other deep learning models is that the recurrent model can handle an infinite amount of input data thanks to the propagation of the recurrent loop. A recurrent neural network propagates not only in space but also in the time dimension. In API recommendation research, several studies have employed LSTM as their recurrent model [48, 19, 7, 49].

In 2016, Gu et al. [7] proposed DeepAPI, an RNN-based sequential model for recommending API method calls. It employs an encoder-decoder structure that converts the text sequence representing the programming question into a list of API method calls. The encoder is a recurrent RNN neural network that transforms the concealed state of each input token into a compressed concealed state C. The decoder then uses another RNN neural network to expand the compressed hidden state C into a list of API method hidden states, which it then maps to a list of API method invocations. Gu et al. [49] introduced DeepCodeSearch (DeepCS) in late 2018, which can be seen as an expansion of DeepAPI. Within the encoder-decoder architecture, DeepCS employs a parallel architecture that models source code and description concurrently. Compared to DeepAPI, this model provides additional information and improves the ranking method from probability to cosine similarity.

Auto Encoder

An autoencoder is a self-supervised model designed to discover dense representations of input data. During training, the objective of an autoencoder is to replicate the input data in the output using a neural network. The middle hidden layer of a trained autoencoder is considered the dense representation of the input data for a given set of input data. SADE is an autoencoder-based API recommendation system for Android programming proposed by Liu et al. [18]. The SADE system suggests API methods with usage patterns similar to a given code snippet. The system extracts information about API usage patterns from the import of API tools and then trains an autoencoder to learn the dense representation of API methods. The trained model serves as an API method input feature extraction model. Given input data, the model generates a ranked list of recommended API method calls.

GNN

A Graph Neural Network (GNN) [135] is a generalization of a neural network for processing graph data. Neural networks are composed of nodes and edges, which can be seen as graph structures. A Convolutional Neural Network (CNN) can also be considered a graph neural network, where the graph data forms a pixel grid. Similarly, in the context of a graph neural network, Recurrent Neural Networks (RNN) can be viewed as a chain structure, where each node represents an input token. In the field of API recommendation, GNN is employed for modeling API method call graphs [36, 72, 16, 14]. Using GNN for modeling call graphs is essential because API method calls are not always sequential, especially when considering execution flow. In scenarios involving concurrency, for instance, certain API call executions do not depend on the completion of a previous API call, allowing two APIs to execute in parallel. The actual execution of API calls forms a graph at runtime. In such cases, GNN is well-suited for processing API call graphs.

Language Models

A language model is a model that learns the structure of a language by analyzing the probability patterns of word sequences in the training corpus. Language models can be built using various approaches, including neural networks or statistical models. Unlike Word2Vec, which focuses on learning synonyms, a language model assigns probabilities to input sequences. Language models find applications in tasks such as sentence vectorization and named entity recognition (NER).

Language models represent the cutting edge of source code modeling techniques. For instance, Yin et al. [27] employ the BERT model for the NER task of constructing an API method knowledge graph. The model identifies named entities and relationships from programming questions on StackOverflow and then uses the extracted entities and relationships to construct the API knowledge graph. Kang et al. [13] propose APIRecX, an API recommendation model based on the Generative Pre-Training (GPT) model. It models GitHub source code fragments and generates an API call list using beam search.

To further bridge the gap between query embedding and code embedding, Wei et al. [6] introduced CLEAR, an API recommendation tool based on contrastive learning. It utilizes the RoBERTa model for natural language modeling, incorporating a contrastive learning objective. This approach entails the model learning to discern the similarity between two provided inputs rather than solely focusing on learning specific characteristics of each classification category. Consistent with previous observations, language models are predominantly employed to capture the natural language aspects of API recommendation queries.

2.7.6 Challenges

Lack of Adoption of Code Models

In Table 2.9, out of 37 studies, only 5 specifically use machine learning models dedicated to code. While general language models can generate code, the significant semantic and syntactic differences between programming languages and natural language prevent these models from matching the performance of code-specific models. Given the accuracy requirements and importance of API recommendation, the use of code models is essential. Although several large language models for code have been proposed in recent years [136] [137], their application in API recommendation remains limited.

Instead, many existing studies train source code models from scratch for source code modeling. However, this practice is inefficient compared to the pre-training and fine-tuning paradigm in natural language processing. Existing studies often resort to repurposing pre-trained deep learning models from the natural language processing field for source code modeling through fine-tuning, despite the scarcity of pre-trained models specifically designed for source code. Researchers should adopt code models for API recommendation studies to improve code comprehension capability and semantic representation.

Table 2.10: Common Evaluation Metrics

No	Metrics	Reference
1	Accuracy	[30, 13, 19, 120, 71, 48, 8, 46, 89, 122]
2	MRR	[115, 21, 110, 107, 120, 8, 46, 108, 89, 112, 88, 5, 22, 109, 122, 6]
3	MAP	[21, 110, 107, 8, 46, 108, 112, 88, 5, 22, 109, 31, 115, ?, 113, 6]
4	BLEU	[48, 121, 119, 7]
5	Hit ratiao	[108, 88, 5, 22, 33]
6	Precision	[43, 15, 116, 57, 115]
7	Recall Rate	[109, 46, 116, 57, 31, 18, 106, 25]
8	NDCG	[22]
9	F1 Score	[116, 57]
10	Success Rate	[111, 42, 33, 105, 107]

Lack of Language Generalizability

This thesis finds that out of the 37 collected articles, 20 studies concentrate on Java API recommendations. Only a limited number of studies are dedicated to other languages, such as Python, JavaScript, and C. No studies can be found on languages like PHP, Go, or Kotlin. Considering that each programming language has its strengths, widely accepted use cases, and community, there may be a gap in API recommendation results between different languages. Therefore, the conclusions drawn from the existing study may not be generalized to other fields. Such an investigation would be valuable, but it remains open for exploration.

2.7.7 Summary and Open Problems

Researchers extract text-based API usage information primarily from official API documentation sources such as the Java JDK Website, Android developer guides, and the Python Standard Library, with each source offering structured and consistent formats for information extraction. The API extraction techniques can be categorized into four categories based on the techniques: Graph analysis, AST parsing, Code parsing, and document parsing. Document parsing and code parsing are the most popular API usage extraction methods, accounting for over 67% of API usage extraction methods.

In API recommendation, three common modeling approaches are utilized: statistical models, machine learning models, and deep learning models. The most popular statistical models include the TF-IDF model and the Indexing Model. Among machine learning models, the Word2Vec model is widely favored. As for deep learning models, popular choices include LSTM, CNN, and BERT models.

2.8 API recommendation Evaluation

The evaluation process constitutes the final and most critical phase in API recommendation tools. Proper evaluation is essential to assess the effectiveness of these approaches. This systematic literature review examines all relevant papers and documents their adopted evaluation methods. Evaluation metrics fall into two primary categories: automatic metrics and human evaluation metrics. The following sections present the evaluation metrics utilized in the collected API recommendation tools.

2.8.1 Common Automatic Metrics

This thesis compared the evaluation metrics used by API recommendation tools. Since API recommendation is generally a problem involving information retrieval, the evaluation metrics employed by the tools are primarily derived from information retrieval metrics. Table 2.10 presents the evaluation metrics utilized by these tools. The next section describes each metric in the table

Accuracy, Precision@K, Recall@K or hit ratio@K, and F1@K

Accuracy, Precision@K, Recall@K, and F1@K are frequently employed metrics for evaluating the performance of a recommendation system. These metrics in API recommendation are typically calculated based on the first K results, where K represents the top K recommendations. Given the answers in a testing set and a model's predictions, the accuracy of a recommendation is the proportion of correctly predicted results relative to the size of the dataset. Precision@K, Recall@K, or hit ratio@K, and F1@K are characterized as follows:

$$Precision@K = \frac{\text{number of top k recommendations that are relevant}}{\text{number of items that are recommended}} \tag{2.1}$$

$$Recall@K = \frac{\text{number of top k recommendations that are relevant}}{\text{number of all relevant items}} \tag{2.2}$$

$$F1@K = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.3}$$

Mean Reciprocal Rank(MRR)

MRR is calculated as the average of the reciprocal rank (RR) across all users. The reciprocal rank, in this context, represents the inverse of the position of the first correctly recommended item. The definition is as follows:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{2.4}$$

Where Q is the length of the test set. MRR is a suitable metric for evaluating API recommendation systems because it is particularly effective when the first correct recommendation holds significant importance. This is typically the case when the first API recommendation result is the most critical API.

Mean Average Precision(MAP)

Mean average precision measures the mean of average precision (AveP) of the test result. The formula of AveP and MAP is defined as the following:

$$AveP@K = \frac{1}{m}\sum_{k=1}^{N}(P(k) \text{ if } k^{th} \text{ item was relevant}) \tag{2.5}$$

$$MAP = \frac{\sum_{q=1}^{Q} \text{AveP}(q)}{Q} \tag{2.6}$$

Where Q is the length of the test set. The MAP metric primarily assesses the comprehensiveness of recommendation results. MAP is a valuable metric in the API recommendation field as it is frequently employed to gauge the completeness of API recommendation results.

Bi-Lingual Evaluation Understudy(BLEU)

BLEU is a widely accepted metric for evaluating the performance of language models in natural language processing. Originally designed to assess the quality of machine-translated text, it has since been extended to evaluate the quality of sequences generated by models. The BLEU score ranges from 0 to 1, indicating the similarity between the translation and the reference translation. In API recommendation, BLEU is employed to measure the similarity between the generated API sequence and the reference sequence, considering both the number of mathematical APIs and the sequence of tokens.

Normalized Discounted cumulative gain (NDCG)

Similar to BLEU, NDCG assesses the quality of a ranked list. The Cumulative Gain (CG) is calculated as the sum of all relevant results within the sequence's length. Discounted Cumulative Gain (DCG) enhances CG by applying a logarithmic function to consider the sequence order. To account for varying sequence lengths and to normalize the score within the range of zero to one, Normalized Discounted Cumulative Gain (NDCG) is used. However, NDCG is less commonly utilized in evaluating API recommendation results, particularly in cases where it is insensitive to false positives.

Success rate

Success rate measures the rate of at least one successful match in the top K results. For example, for a dataset P, the formula for the success rate is defined as:

$$\text{success rate}(p) = \frac{\sum_i^N match_i(p)}{\sum_i^N |G_i|} \times 100 \tag{2.7}$$

2.8.2 Metrics for Human Evaluation

Although human evaluation has been considered critical in API recommendation [50, 30, 5], only a few studies performed Human evaluation studies in their work [7, 87]. This thesis thoroughly reviewed the selected papers and identified five studies that conducted user studies to evaluate the usefulness of their API recommendation tools. Table 2.11 provides a summary of the details of these user studies. It's worth noting that all of these user studies have limitations in terms of the number of participants and tasks. Huang et al. [21] conducted the largest survey in terms of the number of participants, involving 28 students and developers who evaluated the proposed tool and several baseline tools by searching for solutions. Chen et al.[14] conducted a user study to evaluate the effectiveness of their tool, APIRec-CST. They developed an IntelliJ IDEA plugin and asked participants to complete six programming tasks with and without the tool. The researchers recorded the completion time and pass rate of each task and also conducted interviews with the participants. The study revealed two key findings. Firstly, participants suggested that the API recommendation tool should also provide argument examples. Secondly, participants requested a short explanation along with the API recommendation. Similarly, Xie et al.[30] evaluated the effectiveness of their tool, PreMA, by conducting a user study. They asked 12 participants to complete 30 tasks using both PreMA and a baseline approach. For each task, the participants were asked to try queries that generated a list of APIs and continue until they found the APIs that could help them complete the tasks. The difference is that the researchers recorded the accuracy, number of retries, and time taken to complete each task. Gu et al. [7] performed a user study with 30 queries and asked the participant to select the most relevant result based on the recommendation result. They then evaluate the top 5 and top 10 relevancy ratios. The result shows that the Wilcoxon signed-rank p-value result across all participants is less than 0.05, which means a high consensus about the result. Ponzanelli et al. [87] have used a simplified approach to evaluate the usefulness of their tool. Five participants were asked to rate the tools on completeness, prominence, information quantity, and whether they would recommend it to others.

The lack of human evaluation could be due to the difficulty of finding enough qualified participants and the time required for the ethical review process. However, it is still essential to conduct human evaluations to understand the effectiveness of their tool and identify areas for improvement. Without it, the validity of their API recommendations may be compromised, posing a significant threat to the usefulness of API recommendation tools.

Table 2.11: List of Studies Considering User Study

Paper	participants	evaluation method
Gu et al.[7]	2 independent developers	Accuracy
Xie et al.[30]	(2 PhD students and 10 MS students)	Number of Retries, Completion Time
Li et al.[115]	15 developer, 30 student	Understandability, Usability
Huang et al.[21]	28 Java developers	Correctness, Completion Time
Wei et al.[110]	4 PhD students and 10 MS students	Correctness, Completion Time
Wang et al.[107]	less than 10 student	General Feedback

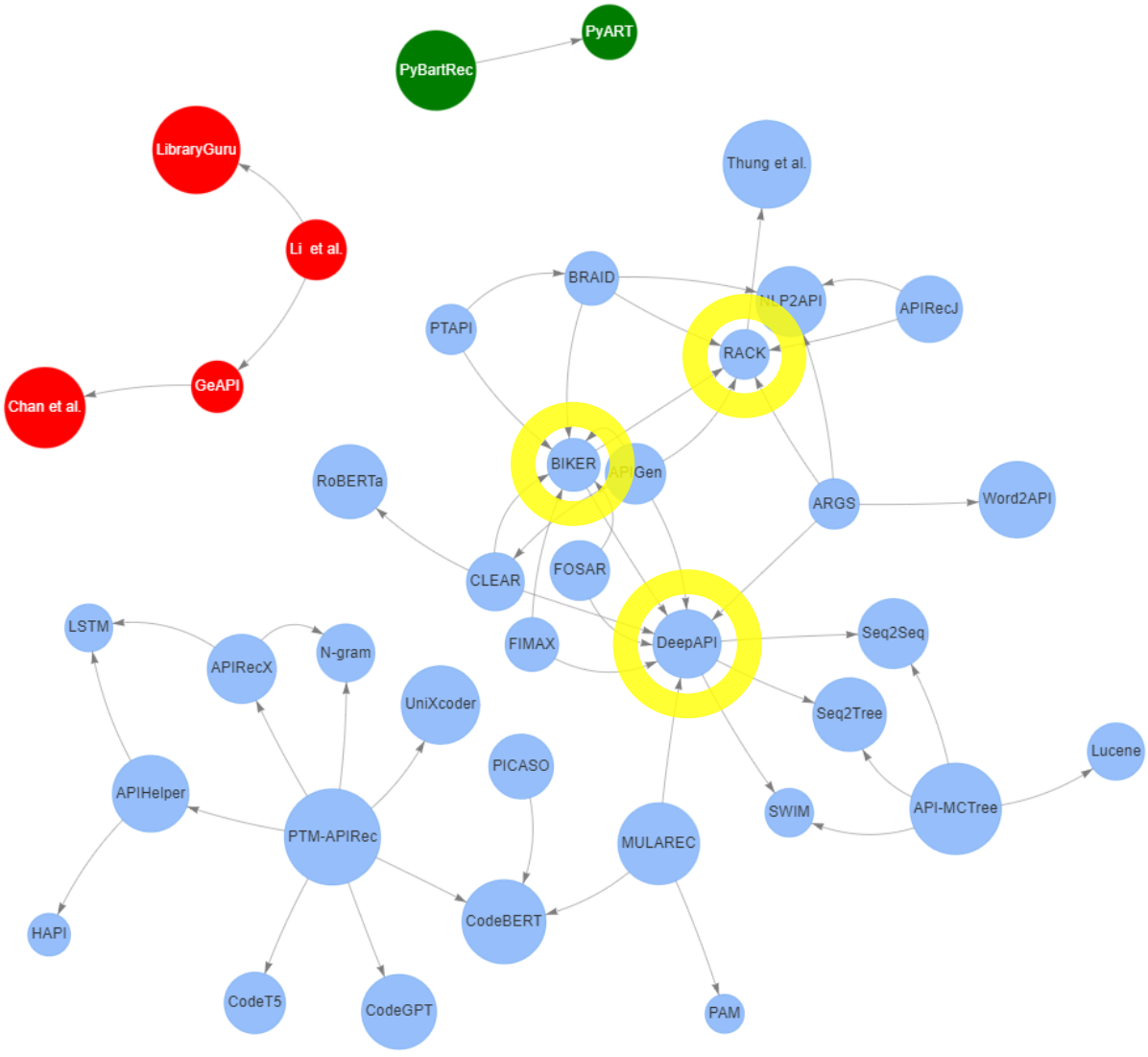

Figure 2.8: The overview of query-based API recommendation.

2.8.3 Consistency Across Metrics

The majority of research papers do not reuse existing datasets but rather collect their own version of the dataset using the same approach as previous studies. There are a few reasons for this. For example, the tool may require a new type of information that existing datasets cannot provide, such as code change history or API context graph. Another common reason is that the API recommendation dataset needs to be updated periodically due to version updates. This ensures that the tool is useful for the latest version of APIs. Two datasets are commonly reused: the Garlan Dataset and the BIKER&RACK dataset. Table 2.8 provides details of research that reuse these datasets. The rows represent the names of the datasets, and the columns represent the names of the tools that reuse these datasets, sorted by performance from worst to best from left to right. The MRR metric is the most consistent and commonly accepted evaluation metric for the evaluation of the two datasets. The table shows that the Garlan Dataset has been reused for five studies, and the BIKER&RACK dataset has been reused for six studies. The popularity of these two datasets can be attributed to their simplicity, comprehensiveness, and accessibility. Both datasets are publicly available and cover a large number of instances and projects. The number of columns and data format are straightforward, making it easy to adapt them for new projects.

Out of the eleven evaluation metrics, Accuracy, MRR, and MAP are the most widely accepted ones for API recommendation evaluations. Precision, Hit ratio, and Recall are also frequently used for this purpose. Human evaluation is still a missing piece of the evaluation metric. Although previous studies have highlighted the importance of human evaluation in verifying the usefulness of API recommendations, few studies have considered human evaluation.

2.9 Conclusion

This chapter provides a comprehensive overview of existing approaches in API recommendation systems, focusing on statistical, machine learning-based, and deep learning models. The approaches range from traditional indexing and TF-IDF techniques to advanced methods like Graph Neural Networks (GNNs) and language models. this thesis's analysis reveals the strengths and limitations of each method, highlighting the evolution of API recommendation techniques.

This thesis have identified several key challenges in the current state of API recommendation research. The most pressing is the limited adoption of code-specific models, which hampers performance in API recommendation tasks due to the distinct semantic and syntactic characteristics of programming languages compared to natural language. Although large language models for code have emerged recently, their application in API recommendation remains minimal, necessitating further exploration and adoption. Moreover, the lack of generalizability of languages across various programming languages restricts the applicability of existing models, with most studies predominantly focusing on Java. Future research

should address this gap to provide more comprehensive and universally applicable API recommendation systems.

The evaluation process is crucial to assess the effectiveness of these tools. Common automatic metrics, such as Accuracy, Precision@K, Recall@K, F1@K, MRR, MAP, BLEU, and NDCG, have been widely adopted, while human evaluations remain underutilized despite their potential to offer valuable insights into practical usability. Expanding user studies with a wider range of participants and tasks will further enhance the reliability and applicability of API recommendation tools.

In conclusion, while significant progress has been made in API recommendation research, addressing the identified challenges will be critical to advancing the field and enabling the development of more effective, generalizable and user-friendly API recommendation systems.

Chapter 3

Recommending API using Supervised Contrastive Learning

3.1 Introduction

Over the past few decades, open-source software development has received extensive attention from the software engineering community. This leads to a tremendous demand for already devised libraries or APIs that facilitate software development and maintenance. Developers often search for existing APIs or code snippets on the Internet to obtain the functions they wish to implement [138].

To help with API search, many automated API recommendation approaches have been proposed and achieved remarkable performance [21, 73, 8, 139, 24, 80, 140]. Most existing approaches leverage the question & answer pairs (Q&A pairs) from Stack Overflow (SO) [141] and API documentation to recommend APIs for a given natural language-described programming task query. In this thesis, for the clarity of writing, a post refers to the title of a single SO post. Query refers to the given input query when searching relevant APIs.

Recently, Huang et al. proposed BIKER [21], a state-of-the-art API recommendation approach that uses word embedding (i.e., word2vec model [142]) to calculate the similarity score between two text descriptions and leverages a query's similarity with both SO posts and API documentation to recommend appropriate APIs for the query. Specifically, given a query, it first uses bags-of-words to represent both the query and SO posts and further use word embedding to calculate the similarity between this query and all candidate SO posts and return the ranked list of posts. Then, it re-ranks the top-50 most similar SO posts by text similarity between this query and the documents of APIs in the posts. Finally, APIs from the re-ranked SO posts are recommended for this query. However, this thesis found two major problems in their work. Table 3.1 shows two examples of BIKER recommendation results corresponding to the two problems.

Table 3.1: Top three similar SO posts recommended by BIKER for the two given example queries. ✓ indicates the ground-truth API and ✗ indicates the recommended API is incorrect.

	Query/SO posts	**API**
First example query	Convert String to Calendar Object in Java	✓ java.time.YearMonth.from
1st	Convert Java Gregorian Calendar to String	✗ java.time.LocalDate.parse
2nd	Convert int to Calendar Object	✗ java.time.LocalDateTime.parse
3rd	Converting String to Date - Java	✗ java.text.DateFormat.format
Second example query	Filereader.read() method not working	✓ java.io.OutputStreamWriter.flush
1st	BufferedReader read() not working	✗ java.io.BufferedReader.read
2nd	when does FileInputStream.read() block?	✗ java.nio.FileInputStream.read
3rd	How to return bytes with randomAccessFile.read() method?	✗ java.io.BufferedReader.read

The first issue with BIKER is its limitation in capturing semantically related sequential information within text descriptions of queries and posts. For instance, consider the real-world query *Convert String to Calendar Object in Java*"[5] presented in Table 3.1. BIKER recommends the API `java.time.LocalDate.parse`" from the most similar post it identified, namely "*Convert Java Gregorian Calendar to String*"[6], which has the opposite intent of the query. BIKER's failure to retrieve the correct answer for this query arises from its use of a bag-of-words model [143] to represent the text descriptions of queries and Stack Overflow posts, which cannot capture the sequential semantic information needed. To adequately represent the semantic-related sequential information in text descriptions, the embedding of queries and Stack Overflow posts needs to be considered in a comprehensive manner rather than as a simple bag of words.

The second issue with BIKER is its difficulty in differentiating the semantic nuances among lexically similar queries. For example, given the query *Filereader.read() method not working*"[7], shown in Table 3.1, BIKER suggests the API `java.io.RandomAccessFile.read`" based on the post *BufferedReader read() not working*"[8]. Despite the query and post being nearly identical, differing primarily by `Filereader`" versus `BufferedReader`," the correct answer is actually `java.io.OutputStreamWriter.flush`." This failure results from the two queries being lexically similar yet semantically distinct. BIKER's word2vec semantic embedding relies on the words' contexts within the text description. However, this example shows that merely using word context is insufficient to distinguish semantic differences in API recommendation tasks. To mitigate this problem, an approach that can better identify semantic differences among queries is needed.

To alleviate the above two problems, this thesis propose CLEAR, a contrastive training-based API recommendation approach. Contrastive training [144] (details are in Section 3.3.1) is a self-supervised model training approach that takes a triplet (S, P, N) as the input, where S corresponds to the original query, P refers to the positive equivalent of S, and N is the

[5]https://stackoverflow.com/questions/5301226/convert-string-to-calendar-object-in-java
[6]https://stackoverflow.com/questions/24741696/convert-java-gregorian-calendar-to-string
[7]https://stackoverflow.com/questions/36427839/filereader-read-method-not-working
[8]https://stackoverflow.com/questions/43190995/bufferedreader-read-not-working

negative one. The training objective is to learn the semantic relationship between queries, i.e., whether two queries are semantically equivalent regardless of their lexical information.

Specifically, the CLEAR first employs a BERT-based model for text embedding, which generates an embedding of the entire sentence of an API query, incorporating sequential information rather than combining individual word embeddings (i.e., bag-of-words). Additionally, CLEAR uses contrastive training to enhance semantic learning, allowing it to capture query semantics by learning which pairs of queries are semantically equivalent. This approach enables CLEAR to recognize queries that are semantically equivalent but lexically distinct through contrastive training.

To evaluate the effectiveness of CLEAR, this study utilizes the dataset from prior work, BIKER [21]. Three test sets were derived from BIKER's dataset: (1) the original BIKER test dataset, (2) a randomly selected collection of 1,000 Stack Overflow (SO) posts, and (3) an additional set of 1,000 SO posts containing multiple APIs in their answers. The third set was created to assess performance in multi-API scenarios, as approximately 10% of SO posts involve multiple APIs. CLEAR was trained using a dataset that explicitly excluded these test cases.

The experimental results demonstrate that CLEAR significantly outperforms state-of-the-art baselines at both method and class levels across all three test sets. A case study evaluating CLEAR against the latest Stack Overflow posts further confirms the approach's effectiveness and practical value.

This thesis makes the following contributions:

- This thesis proposes a novel API recommendation approach, CLEAR, which embeds API queries by encoding the entire query sentence using a BERT-based model and applies contrastive training to learn query semantics. To the best of this thesis's knowledge, CLEAR is the first API recommendation approach capable of learning the semantics of queries independently of specific programming tasks.

- This thesis evaluates CLEAR using three large-scale test datasets, including data from previous studies, 1,000 randomly selected SO posts, and 1,000 randomly selected SO posts with multi-API answers. Experimental results confirm that CLEAR outperforms state-of-the-art baselines.

- This thesis conducts a case study on the latest SO posts to assess the performance of CLEAR , with results suggesting CLEAR's practical value.

- This thesis releases the source code for CLEAR and the experimental dataset to aid researchers in replicating and extending this thesis[9].

The rest of this chapters is structured as follows. Section 3.2 describes the background of this study. Section 3.3 presents the framework of the proposed CLEAR. Section 3.5 introduces experimental design, baseline and research questions. Section 3.6 analyzes the experiment results.

[9]Reproduction package link: http://doi.org/10.5281/zenodo.4712643

3.2 Background

3.2.1 Bag-of-words

Bag-of-words is a language representation technique for natural language process and information retrieval [143]. Bag-of-words convert a sentence as a bag of its word tokens disregarding word order. Bag-of-words is often used as a feature extraction tool that generates spars integer vector form sentences. Many existing API recommendation approaches adopt bag-of-words to represent queries or SO posts [21, 8].

3.2.2 Language Embedding

Language embedding technique is a method for converting words or sentences into numerical vectors. The purpose of language embedding is to obtain a numerical output that represents the meaning of the input sequence of natural language tokens. There are existing studies of language embedding on both word-level [142, 145] and sentence-level [146, 132, 147].

Word2vec [117] is a classic word embedding technique. The word2vec model converts words into vectors by learning the contextual knowledge of the words. In the vector space of the trained model, the vector of similar words is closer than the vectors of unrelated words [148, 149]. Such a model can represent semantic relations at the word level. Please note that the linear combination of word vectors, i.e., using the mean value of the word vectors as the sentence vector may not accurately represent the semantics of the sentence [150].

BERT [132] is a pre-trained language model neural network based on transformer units. It employs a transformer layer with a multi-head attention mechanism for feature extraction, followed by a regression function to generate the final output. The original BERT model is trained through two tasks: predicting missing tokens in sentences and determining whether one sentence follows another. For single sentence embedding extraction, the sentence is provided as input to the model, with the output vector serving as the sentence embedding. This work utilizes two variations of BERT models as embedding extraction layers: Distil-RoBERTa [130] for the filtering model and Tiny-BERT [151] for the re-ranking model. These models were selected because they represent state-of-the-art approaches in semantic search and re-ranking, respectively [152]. Both BERT-based models operate similarly, taking a query text as input and producing a vector that serves as the numerical representation of the query.

3.2.3 Contrastive Training

Contrastive training [144] is a deep neural network training process that takes paired sentences as input and uses the similarity in the paired sentences as labels. The training goal is to learn the relationship between sentences, i.e., whether two sentences are semantically similar regardless of their lexical similarity. Hoffer et al. [153] proposed the triplet network for contrast training. It requires triplets as input, which contains a sentence, a sentence with the same label as the positive sample, and a sentence with a different label as the negative sample. The first layer of the triple network is three identical deep neural network models

for feature extraction of input sentences. The feature extraction layer can also be replaced with other models or algorithms. The second layer of the triplet network is a loss function based on the cosine distance operator. The purpose of the loss function is to minimize the distance between similar sentences and maximize the distance between unrelated sentences. CLEAR uses contrastive training for training the filtering model.

3.2.4 Joint Embedding Training

Devlin et al. [132] proposed a joint embedding training process in BERT for the Next Sentence Prediction (NSP) tasks. Joint embedding training takes paired sentences as input. Concatenating two sentences as input enables the network structure to extract deeper connection information Through this process, the model can learn the relationship between the two sentences more deeply. CLEAR uses joint embedding training for training the re-ranking model.

3.3 Approach

Figure 3.1 shows the pipeline of this thesis's approach, which consists of two parts: model building (section 3.3.1) and search APIs (section 3.3.2). The model building process contains four steps, which are Q&A pairs collection (section 3.3.1), triplets construction (section 3.3.1), the contrastive training for building the filtering model (section 3.3.1) and the joint embedding training for building the re-ranking model (section 3.3.1).

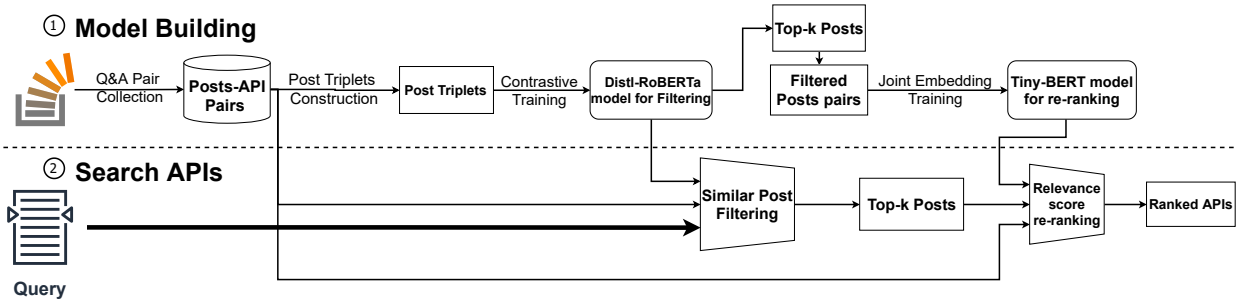

Figure 3.1: Overview of CLEAR

3.3.1 Model Building

Q&A Pair Collection

To compare with the state of the art, this thesis reuses the dataset provided by BIKER [21], which contains 1,347,980 Stack Overflow posts that correspond to Java programming questions. This thesis has also reused the question-API pair extraction method used in BIKER for data extraction. There are 33,871 question-answer pairs successfully recognized, which is consistent with the dataset used in BIKER's experiments.

Post Triplets Construction

The format of the training data used in the contrastive training process is different from the traditional natural language processing tasks, e.g., sentiment analysis, where the input sentences and the outputs are the labels. Contrastive training requires triplets as inputs [153]. Every single triplet is a combination of three posts, which are an input question S, a positive sample question P that is semantically equivalent to S, and a negative sample question N that is not related to S and P. Therefore, the training corpus needs to be converted to triplets. For example, given an input question "*Java string split with multiple delimeters*", the triplet (S, P, N) can be ("*Java string split with multiple delimeters*","*How to split a path using StringTokenizer?*","*How to load a file across the network and handle it as a String*"). Algorithm 1 shows the process of generating training triplets.

Algorithm 1 Triplets Generator

```
Result: Tuple list of element (S, P, N)
def getTriplets(p: int, n: int, T: list<question,
 answer>):
    result_list = new list() // initialize empty
     result list
     for item in T do
        S = item[0] //question sentence
         answer = item[1]
         T_P=get_semantic_equivalent_subset(T,
         answer)
         T_N = set(T) - set(T_positive)
         P_list = random_sample(T_P, p)
         N_list = random_sample(T_N, n)
         for item P in P_list do
            for item N in N_list do
               result_list.append(S, P, N)
            end
        end
    end
    return result_list
```

The algorithm of triplets generation requires three parameters. m is the minimum number of occurrences for each API, p is the number of positive samples per training instance, and n is the number of negative sampling per training instance. When generating the triplets, each question needs to be paired with positive and negative samples. The data balancing between positive and negative samples is important and necessary in contrastive training, different configurations may impact the result significantly [154, 155].

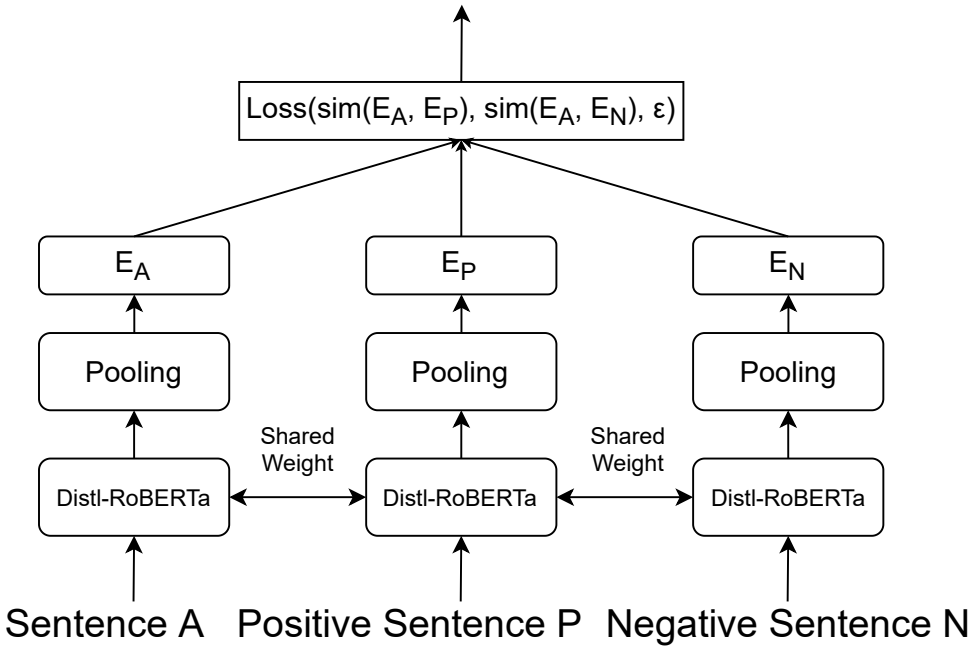

Figure 3.2: Filtering model architecture

Without sampling, the available candidates for negative samples are the entire training sample minus the relevant samples. This means that the total number of training triplets is equal to $T * T\_P * (T - T\_P)$ where T is the total number of training samples and T_P is the total number of relevant passages. This generates too many negative samples for each instance and breaks the balance between positive and negative samples. Therefore, following the existing studies [153, 156], the data augmentation process was implemented by first limiting the minimum occurrence of APIs to five, then randomly sampling the positive and negative samples that exceed the maximum number of occurrences. To determine the optimal configuration, a grid search was conducted on a list of ratios, with results compared to select the best experimental configuration (details provided in Section 3.5.2). Random sampling was applied to APIs exceeding p to limit the number of positive samples per instance. Please note that the random sampling is performed per instance, which means that the positive samples provided to training instances can be different. The reason for this is to mitigate over-fitting in the training process and improve generalizability.

Filtering Model Building with Contrastive Training

As mentioned in Section 3.2, This Thesis uses a triplet network for the filtering model. Figure 3.2 shows the architecture of the filtering model used in this thesis. To capture the semantic-related sequential information, the Distill-RoBERTa [130] model was used as the base model for sentence embedding because it is the state-of-the-art pre-trained language

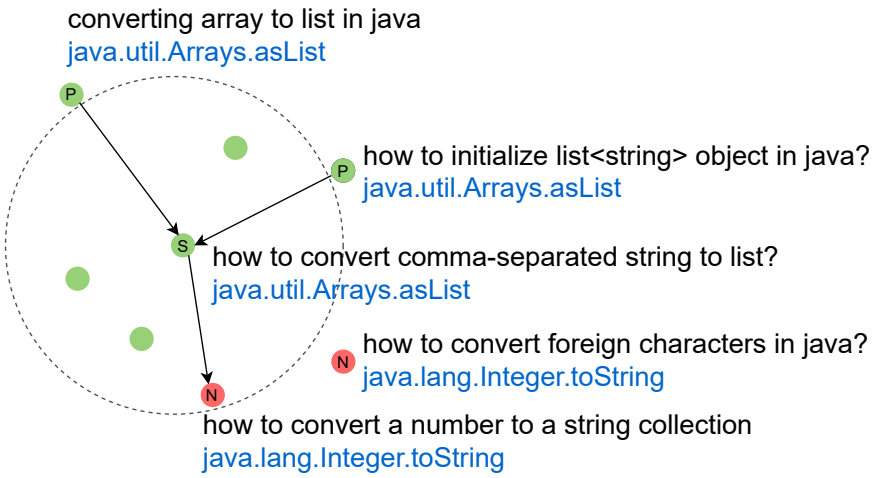

Figure 3.3: Contrastive training for a single API

model for semantic embedding tasks. The Distill-Roberta model was fine-tuned by providing the triplets with contrastive training to the model.

In contrastive training, for a given question S, the distance from S to the semantically equivalent question P should be reduced, and the distance from S to the unrelated question N should be increased. Figure 3.3 shows an illustration of such optimization. In the Figure, green points are the embeddings with the positive API "`Arrays.asList`" and the red points are the embeddings with negative API "`java.lang.Integer.toString`". The center green point S plays the role of an anchor, the positive samples are pulled towards the anchor and the negative samples are pushed away from the anchor.

The training objective is to fine-tune the network so that the distance between the question S and the positive question P is closer than the distance between the question S and the negative question N. Formally, the training objective is to minimize the following function:

$$max(||E_s - E_p|| - ||E_s - E_n|| + \epsilon, 0) \tag{3.1}$$

where E_s, E_p, E_n are the sentence embeddings of question S, P, and N. ϵ is the margin of the distance between S and N. By default, ϵ is set to 1, which means that the cosine distance between the question and the irrelevant question should be 1.

As a result of this step, the filter model returns a list of top candidates $C_1, C_2, C_3, ..., C_n$ with its similarity score $Sim_1, Sim_2, Sim_3, ..., Sim_n$ to the given question S.

Re-ranking Model Building with Joint Embedding Training

Following existing studies [21, 152], a re-ranking model is applied after the filtering to further improve the performance. The objective of the filtering model is to select the top-k results from the entire search space, while the re-ranking model is to optimize the ranking of the

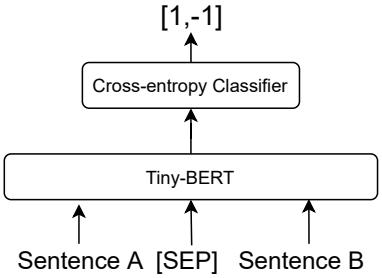

Figure 3.4: Joint embedding training

selected top-k results. For semantic embedding re-ranking tasks, the Tiny-BERT model [151] was selected as the base model, representing a state-of-the-art distilled pre-trained model for this purpose. The model was subsequently fine-tuned using joint embedding training. Figure 3.4 illustrates the architecture of the re-ranking model. The BERT [132] architecture includes a special $[SEP]$ token, enabling the concatenation of two sentences as input. In joint embedding training, $[SEP]$ is used to identify the end of the first sentence. The process of joint embedding training is fine-tuning the model with pairs of questions to the target that if given two semantically equivalent sentences, the model returns 1; otherwise, it returns 0. For each query sentence pair, the first query in the pair is the sentence S from the training corpus, and the second query sentence is one of the results C_n from the filtering model given input S. For each input S, k pairs of (S, C_n) and label y are generated for joint embedding training. For example, with k equal to 50, there are 50 combinations of query pairs for each sentence in the training corpus. The loss function used for joint embedding training is the classic cross-entropy loss function (i.e., $Loss$):

$$Loss = -(y * log(\hat{y}) + (1 - y) * log(1 - \hat{y})) \tag{3.2}$$

where y indicates whether sentences S and C_n are semantically equivalent, and $\hat{y}$ is the prediction of the re-ranking model.

3.3.2 Search APIs

Following the existing studies [156, 153, 21], cosine similarity was applied as a similarity measurement metric. Given a query Q, the filtering model returns a list of candidates sorted by cosine similarity score of the question sentence embedding. It returns the top K most relevant posts from the API search space and their corresponding API methods as candidates. Then, in the re-ranking phase, the re-ranking model calculates the relevance between Q and the top K candidates and returns the softmax result as the relevance score between Q and each candidate. The final outcome is a list of API methods ranked by relevance score.

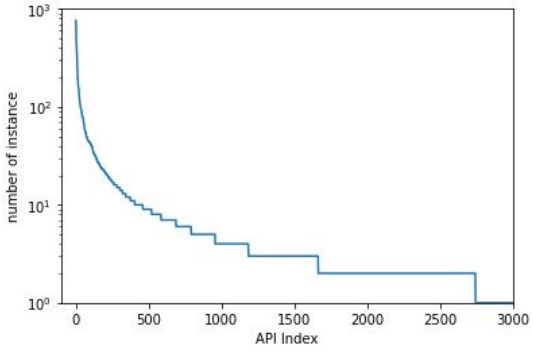

Figure 3.5: Occurrence distribution of APIs in BIKER dataset

3.3.3 Adaptation for Class-Level Recommendation

CLEAR recommends APIs at the method level by default. It can also be easily adapted to class-level recommendations as well. In the case of API class searching, the method name of the candidate API was removed to adjust the candidate API to the class level.

3.4 Evaluation

3.5 Experiment Design

3.5.1 Dataset

BIKER's training dataset contains 33,872 Q&A pairs [21]. The contrastive training process requires a minimum number of instances for each API, necessitating filtering of the training data to ensure sufficient samples for each API. Analysis of the training dataset reveals that the occurrence of training examples for each API label follows a power-law distribution [157], indicating data imbalance. Figure 3.5 displays the frequency curve of unique APIs sorted by occurrence count, with the x-axis representing the sorted index of unique APIs and the y-axis showing their occurrence frequency. The chart demonstrates that only approximately 1,000 APIs appear more than five times. The data augmentation process described in Section 3.3.1 was applied with the minimum occurrence threshold set to five. This process resulted in a final training dataset containing 21,479 instances, comprising 955 unique API answers, including 185 unique Multi-API answers. With both positive and negative sampling parameters set to 10 (detailed in Section 3.5.2), the generated training triplets totaled 370,304 pairs.

When searching for the relevant APIs, all test data was removed from the search space to prevent the experiment from data contamination. The final API Q&A search space for searching relevant APIs contains 20,479 instances.

Table 3.2: Accuracy@1 on different (P)ositive sampling and (N)egative sampling settings

P\N	1	3	5	**10**	15
1	0.004	0.032	0.027	0.036	0.027
3	0.184	0.332	0.392	0.463	0.416
5	0.376	0.512	0.552	0.766	0.76
10	0.524	0.704	0.652	**0.828**	0.784
15	0.624	0.684	0.736	0.72	0.784

3.5.2 Experiment Settings

This thesis use Google Colab [158] professional version for fine-tuning the models. Two Intel Xeon 2.20GHz CPUs with 5 GB cache were used for the experiment. The GPU resource used is an NVIDIA V100 graphics card with 13G memory. The filtering model and the re-ranking model were fine-tuned for five epochs each, and then the model with the best performance on the validation set was selected.

The triplets generation algorithm in CLEAR has two parameters, i.e., the number of positive samples (p) and the number of negative samples(n), which could affect the performance of CLEAR. To determine the optimal values for these two parameters, a grid search is conducted over five discrete values for p and n: 1, 3, 5, 10, and 15. This results in a total of 25 model configurations.

Due to the computational cost of fine-tuning on the full dataset, the grid search is performed using a quarter of the training data. The filtering model is trained until full convergence or for a maximum of 5 epochs to ensure thorough training. A fixed random seed ensures consistency in the random sampling of positive and negative samples across all models. Following existing studies [159, 156], the *accuracy*@1 was used as the metric for parameter tuning, which is calculated as $\#(first\ match\ is\ correct)/\#(test\ instances)$.

Table 3.2 shows the result of *accuracy*@1 based on different parameter settings; the row and column indices are the number of positive and negative samples, respectively. Overall, the performance of CLEAR increases with the increase of positive and negative samples, and the performance of CLEAR reaches the peak at the point where the number of positive and negative samples is equal to 10. Thus, 10 positive samples were set and 10 negative samples for each training instance when training CLEAR in this thesis's experiments. In the case that there are fewer than 10 positive and negative samples, all positive and negative samples are included.

3.5.3 Evaluation Queries

This thesis prepared three different datasets for evaluating the performance of CLEAR and the baseline approaches.

- **BIKER test data:** contains 413 manually selected and verified SO queries with API answers from BIKER's test dataset.

- **Random test data:** contains 1K random selected SO queries with API answers from the BIKER training corpus.
- **Multi-API test data:** contains 1K random SO queries with multi-API answers only from the BIKER training corpus.

3.5.4 Baselines

This thesis compared CLEAR with BIKER [21] and RACK [8], which are two state-of-the-art API recommendation techniques. To show the impact of contrastive training, this thesis also introduces a variant of the filtering model without adopting the contrastive training, which is the pre-trained Distil-RoBERTa model.

Baseline1 (BIKER-filtering) [21]: is the first stage of the BIKER model. For a given query, BIKER uses a mixture of TF-IDF and a trained Word2vec model to calculate the sentence embedding of that query. Then it compares the cosine similarity of the query sentence embedding and the sentence embedding of every question in the API search space. The result is a list of the 50 most similar SO questions ranked by cosine similarity.

Baseline2 (BIKER-complete) [21]: is the complete BIKER model, which leverages both the SO Q&A pairs and the official Java API documents' information for API searching. Taking the result of the BIKER-filtering as input, BIKER-complete re-ranks the top 50 results by comparing the sentence embedding of the top 50 results and the sentence embedding of the corresponding official API document description.

Baseline3 (RACK) [8]: is a keyword-API mapping system that recommends APIs by matching keywords from the query. The keyword API is constructed by mining the statistical relationship between the SO questions and the accepted answers of questions. Please note that RACK only recommends API at the class level.

Baseline4 (Pre-trained Distil-RoBERTa filtering): is the pre-trained Distil-RoBERTa model. This thesis compares CLEAR-filtering with Distil-RoBERTa to explore the performance increase introduced by contrastive learning. In this process, the same pre-trained Distil-RoBERTa model as used in CLEAR is employed.

Baseline5 (CLEAR-filtering): Since CLEAR has two steps, filtering, and re-ranking, the filtering model was separated from the re-ranking model to show the performance increase introduced by joint embedding training.

3.5.5 Performance Measures

Following existing studies [21], this study use Mean reciprocal rank(MRR) [160, 161], Mean average precision(MAP) [162], $Precision@k$, and $Recall@k$, to evaluate the performance of API recommendation approaches. MRR and MAP are the widely accepted measurements for information retrieval. MRR measures the effort needed to find the first correct answer in the recommended list, and MAP considers the ranks of all correct answers. This thesis also evaluate the performance with $Precision@k$ and $Recall@k$, where k can be 1, 3, 5, and 10.

For the search result of a query, precision and recall can be defined as follows:

$$Precision@k = \frac{\#(relevant\ items\ retrieved)@k}{\#(retrieved\ items)} \tag{3.3}$$

$$Recall@k = \frac{\#(relevant\ items\ retrieved)@k}{\#(relevant\ items)} \tag{3.4}$$

where the $\#(relevant\ items\ retrieved)$ refers to the number of correctly recommended API, the $\#(retrieved\,items)$ refers to the number of total retrieved APIs, and the $\#(relevant\,items)$ refers to the number of APIs in the answers of the queries.

3.5.6 Research Questions

To evaluate the performance of CLEAR, this thesis have designed experiments to answer the following research questions:

RQ1: *How effective is CLEAR compare with existing API recommendation baselines at method level?*

RQ2: *How effective is CLEAR compare with existing API recommendation baselines at class level?*

RQ3: *How does random sampling affect the performance of CLEAR?*

RQ1 and RQ2 investigate the performance of CLEAR on method and class-level API recommendation tasks. To further demonstrate its advantages, the performance is compared with five state-of-the-art baselines (details in Section 3.5.4). RQ3 explores the impact of the random sampling process in the triplet generation algorithm (details in Section 3.3.1) on the performance of CLEAR.

3.6 Result Analysis

3.6.1 RQ1: Effectiveness of CLEAR at Method Level

Experimental Method. o answer this research question, CLEAR is compared with the baselines listed in Section 3.5.4, excluding RACK, as RACK recommends APIs at the class level only.

For comparison with BIKER, the replication package of BIKER[10] is used to conduct experiments.

Since both BIKER and CLEAR consist of two steps (filtering and re-ranking), an ablation study is performed by removing the re-ranking model from both approaches. This evaluates the performance of BIKER and CLEAR in the filtering step, referred to as BIKER-filtering" and CLEAR-filtering," respectively.

[10]https://www.dropbox.com/s/fr4gdbyfn58ytm8/BIKER.zip?dl=0

Table 3.3: Performance comparison at method-level test data(RQ1)

Method-level			BIKER-filtering	BIKER-complete	Distil-RoBERTa	CLEAR-filtering	CLEAR-complete
BIKER	MRR		0.4318	0.6225	0.4098	0.6319	**0.7551**
	MAP		0.4260	0.6175	0.4088	0.6398	**0.7655**
	Precision	P@1	0.2777	0.4642	0.2341	0.4206	**0.4682**
		P@3	0.2328	0.2486	0.2632	0.5132	**0.5502**
		P@5	0.2071	0.1698	0.2563	0.5206	**0.5531**
		P@10	0.1928	0.0956	0.2305	0.5138	**0.5563**
	Recall	R@1	0.2678	0.4503	0.2321	0.4087	**0.6309**
		R@3	0.5019	0.7142	0.4980	0.6607	**0.7638**
		R@5	0.5972	**0.8134**	0.6130	0.7301	0.7956
		R@10	0.7440	**0.9166**	0.7182	0.7738	0.8551
Random	MRR		0.2448	0.2813	0.2912	0.7573	**0.8042**
	MAP		0.2357	0.2724	0.2855	0.7612	**0.8040**
	Precision	P@1	0.1420	0.1740	0.1940	0.6680	**0.7220**
		P@3	0.1266	0.1103	0.1746	0.6669	**0.7080**
		P@5	0.1160	0.0830	0.1673	0.6495	**0.6909**
		P@10	0.1074	0.0546	0.1524	0.6233	**0.6727**
	Recall	R@1	0.1298	0.1620	0.1783	0.6383	**0.6906**
		R@3	0.2673	0.3011	0.3093	0.8078	**0.8446**
		R@5	0.3298	0.3791	0.3791	0.8523	**0.8840**
		R@10	0.4418	0.4976	0.4724	0.8954	**0.9328**
Multi-API	MRR		0.2296	0.2879	0.2988	**0.6495**	0.5876
	MAP		0.2212	0.2804	0.2895	**0.6392**	0.5755
	Precision	P@1	0.1280	0.1860	0.1970	0.6770	**0.7200**
		P@3	0.1166	0.1156	0.1766	0.6489	**0.7009**
		P@5	0.1162	0.0850	0.1692	0.6365	**0.6898**
		P@10	0.1120	0.0542	0.1585	0.6183	**0.6647**
	Recall	R@1	0.1155	0.1703	0.1800	0.6406	**0.6846**
		R@3	0.2440	0.3126	0.3183	0.7806	**0.8306**
		R@5	0.3188	0.3795	0.3891	0.8335	**0.8758**
		R@10	0.4443	0.4856	0.4979	0.8793	**0.9098**

To ensure a fair comparison with BIKER, an API is considered correct if it matches any of the APIs in cases involving multi-API answers.

Results. Table 3.3 shows the result of CLEAR compared with the other baselines.

As shown in the Table 3.3, overall CLEAR outperforms BIKER on both the filtering stage and re-ranking stage. In the evaluation on BIKER test data, the *recall*@1 of CLEAR-complete is 0.6309, indicating that there is at least one right answer in the first candidates in 63.09% cases. Comparing BIKER-filtering model and CLEAR-filtering model, CLEAR-filtering model outperforms BIKER-filtering model by 46.43% and 50.18% on MAR and MAP. In terms of precision and recall, CLEAR-filtering model improves the *precision*@$1, 3, 5, 10$ by 51.45%, 120.44%, 151.37%, 166.49%, *recall*@$1, 3, 5, 10$ by 52.61%, 31.63%, 22.25%, 4.005% respectively.

On the random test data, CLEAR-complete model outperforms BIKER-complete model in all the measurements. Comparing to BIKER-complete model, CLEAR-complete model improves by 185.88% on MRR, 195.15% on MAP, 314.94%, 541.88% 732.24% 1132.05% on *precision*@$1, 3, 5, 10$, and 326.29%, 180.50%, 133.18%, 87.45% on *recall*@$1, 3, 5, 10$ respectively.

Table 3.4: Performance comparison at class-level (RQ2)

Class-level			BIKER-filtering	BIKER-complete	RACK	Distil-RoBERTa	CLEAR-filtering	CLEAR-complete
BIKER test data	MRR		0.6397	0.8138	0.3047	0.5761	0.7059	**0.8765**
	MAP		0.6343	0.8138	0.3001	0.5769	0.7156	**0.8906**
	Precision	P@1	0.2777	0.4642	0.2420	0.3690	0.7777	**0.8134**
		P@3	0.2328	0.2486	0.1203	0.4404	0.7513	**0.8015**
		P@5	0.2071	0.1698	0.0777	0.4269	0.7380	**0.7817**
		P@10	0.1928	0.0956	0.0420	0.4095	0.7182	**0.7496**
	Recall	R@1	0.4623	0.6865	0.2361	0.3611	0.5436	**0.8095**
		R@3	0.7559	**0.9067**	0.3472	0.7202	0.8253	0.8988
		R@5	0.8531	**0.9563**	0.3750	0.8214	0.8750	0.9186
		R@10	0.9424	**0.9880**	0.4067	0.9226	0.8988	0.9345
Random test data	MRR		0.4060	0.4515	0.2343	0.4426	0.8467	**0.8749**
	MAP		0.3961	0.4408	0.2207	0.4410	0.8536	**0.8796**
	Precision	P@1	0.1420	0.1740	0.1520	0.3030	0.7800	**0.8100**
		P@3	0.1266	0.1103	0.0989	0.2976	0.7719	**0.8093**
		P@5	0.1160	0.0830	0.0722	0.2925	0.7611	**0.7989**
		P@10	0.1074	0.0546	0.0431	0.2810	0.7400	**0.7809**
	Recall	R@1	0.2473	0.3103	0.1395	0.2833	0.7473	**0.7751**
		R@3	0.4573	0.4881	0.2728	0.4988	0.8806	**0.9113**
		R@5	0.5568	0.5571	0.3308	0.5908	0.9133	**0.9403**
		R@10	0.6633	0.6880	0.3968	0.7063	0.9395	**0.9606**
Multi-API test data	MRR		0.3829	0.4458	0.2511	0.4351	**0.7702**	0.7215
	MAP		0.3763	0.4371	0.2406	0.4288	**0.7684**	0.7147
	Precision	P@1	0.1280	0.1860	0.1620	0.3040	0.7720	**0.7990**
		P@3	0.1166	0.1156	0.1069	0.2843	0.7513	**0.7933**
		P@5	0.1162	0.0850	0.0758	0.2803	0.7415	**0.7891**
		P@10	0.1120	0.0542	0.0448	0.2698	0.7312	**0.7696**
	Recall	R@1	0.2263	0.3048	0.1525	0.2811	0.7340	**0.7610**
		R@3	0.4341	0.4901	0.3003	0.4830	0.8458	**0.8825**
		R@5	0.5298	0.5620	0.3548	0.5756	0.8853	**0.9223**
		R@10	0.6688	0.6668	0.4208	0.6905	0.9226	**0.9458**

On multi-API test data, CLEAR-complete mode outperforms BIKER-complete mode by 104.09% on MRR and 105.24% on MAP. In terms of precision and recall, CLEAR-complete improves the *precision*@$1, 3, 5, 10$ by 287.09%, 506.31%, 711.52%, 1126.38% and *recall*@$1, 3, 5, 10$ by 301.99%, 165.70%, 130.77%, 87.35% respectively.

Comparing to the Distil-RoBERTa model, CLEAR filtering model achieves better performance on all three test datasets, which indicates that contrastive learning can help increase the model's performance.

This thesis has also conducted the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of CLEAR and baselines. The test result suggests that CLEAR achieves significantly better performance than baseline approaches.

CLEAR significantly outperforms the state-of-the-art baselines at method-level API recommendation, and CLEAR's performance remains stable across different test datasets.

3.6.2 RQ2: Effectiveness of CLEAR at Class Level

Experimental Method. This research question is addressed by applying identical evaluation methodology to BIKER, RACK, and CLEAR . The analysis utilizes the same three test

Table 3.5: Impact of triplets random sampling to model performance

Metric	**MRR**	**MAP**
Average Error	0.85%	0.84%
Coefficient of Variation (CV)	0.004	0.011

datasets, with API methods removed, to enable class-level API answer comparisons. For the RACK comparison, experiments employ RACK's replication package[11].

Results. Table 3.4 shows the result of CLEAR compared with the other baseline approaches at the class level. On BIKER test data, the recall@1 of CLEAR-complete is 80.95%, indicating that there is at least one right answer in the top three candidates in 80.95% cases. Comparing CLEAR-complete model with RACK, CLEAR-complete model outperforms RACK by 187.65% in MRR and 196.76% in MAP. In terms of precision and recall, CLEAR-complete improves the *precision*@$1, 3, 5, 10$ by 236.11%, 566.25%, 906.04%, 1684.76% and *recall*@1 by 242.86%, 158.87%, 144.96%, 129.77% respectively. Comparing CLEAR-complete model with BIKER-complete model, CLEAR-complete model outperforms BIKER-complete model by 7.70% in MRR and 9.43% in MAP.

On the random test data, CLEAR outperforms RACK and BIKER in all the measurements. Comparing CLEAR-complete model with RACK, CLEAR-complete outperforms RACK by 273.41% in MRR, 298.55% in MAP, 432.89% in precision@1, and 455.62% in Recall@1.

On the multiple-API test data, CLEAR-complete model outperforms RACK by 187.33% in MRR, 197.04% in MAP. In terms of precision and recall, CLEAR-complete outperforms RACK the *precision*@$1, 3, 5, 10$ by 393.20%, 642.09%, 941.02%, 1617.85% and *recall*@$1, 3, 5, 10$ by 399.01%, 193.87%, 159.94%, 124.76% respectively.

The Wilcoxon signed-rank test ($p < 0.05$) also suggests that CLEAR achieves significantly better performance than the baseline approaches.

CLEAR significantly outperforms the state-of-the-art baselines at class-level API recommendation and CLEAR's performance remains stable across the three test datasets.

3.6.3 RQ3: Impact of Random Sampling

Experimental Method. In CLEAR's triplet generation process, queries containing more than 10 positive or negative samples result in a random selection of 10 samples from each category. To evaluate the impact of random sampling on CLEAR's performance, the triplet generation process was repeated 10 times. Note that fine-tuning the model with complete training triplets requires significant computational resources; consequently, this experiment utilizes a subset of 92,928 training pairs (representing one quarter of the full training triplets).

Result. Table 3.5 shows the impact of random sampling on the performance of CLEAR measured by the Average Error and Coefficient of Variation(CV). As shown in the table, the

[11]https://github.com/masud-technope/RACK-Replication-Package

Table 3.6: Top similar questions recommended by FIMAX for the two example queries. ✓ indicates the ground-truth API and ✗ indicates the recommended API is incorrect.

	Question	**API answers**
input query	How to convert DateFormat "Fri Jan 08 13:48:16 GMT+05:30 2021" to java.sql.Date	✓java.text.SimpleDateFormat.parse
1st	**How to parse "Thu Aug 04 00:00:00 IST 2011" to "04-08-2011"?**	✓java.text.SimpleDateFormat.parse
2nd	Converting "2010-02-15T20:05:28.000Z" in GMT format using Java	✗java.text.SimpleDateFormat.parse
3rd	Convert String date into java.util.Date in the dd/MM/yyyy format	✗java.text.DateFormat.format
4th	Date format and the hour is always 12:00:00.000	✗java.time.Instant.parse
5th	Java Calendar: getting time for the timezone	✗java.util.Date.toString
input query	How to retrive value from property file which are present outside of the app	✓java.util.Properties.load
1st	**How to close the fileInputStream while reading the property file**	✓java.util.Properties.load
2nd	Using Maven properties to connect to a database	✗java.lang.System.getProperty
3rd	Why do we need Properties class in java?	✗java.util.Properties.load
4th	Issue reading a file path from a Properties file	✗java.util.Properties.store
5th	Parameterizing Java properties file at application launch	✗java.util.Properties.load

average error on MRR is 0.85%, indicating that the difference of MRR introduced by random sampling between different runs is 0.85% on average. The Coefficient of Variation is calculated by $CV = \sigma/\mu$ [163], where σ is the standard deviation and μ is the mean. The CV of this thesis's result suggests that the difference introduced by random sampling is negligible.

The impact of random sampling on the performance of CLEAR is negligible, which shows the robustness of CLEAR.

3.7 Discussion

3.7.1 Why CLEAR Outperforms Existing Baselines?

To investigate CLEAR 's superior performance compared to the baselines in Section 3.5.4, the API search space embeddings are visualized before and after contrastive training.

The analysis employs Uniform Manifold Approximation and Projection (UMAP) [164] for dimensionality reduction of sentence embeddings to two dimensions. These embeddings are then clustered and colored using Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [165], an unsupervised classification approach.

Figure 3.6 upper graph shows the sentence embedding visualization of the training samples on the model before contrastive training, in which the points represent the sentence embedding vectors in two-dimensional space. The color of the points indicates the APIs. The visualization shows that the majority of the APIs are mixed, and the boundary of each APIs is not clear. This graph shows clearly that it is very hard to draw the decision boundary for different clusters in the model before contrastive training. Since the training target of contrastive training is to minimize the distance between semantically equivalent sentences and maximize the distance between the irrelevant sentence, the margin between clusters should be larger and clearer after training.

To support the above hypothesis, the same visualization approach is applied to the fine-tuned model after contrastive training. Figure 3.6 (lower graph) displays the sentence

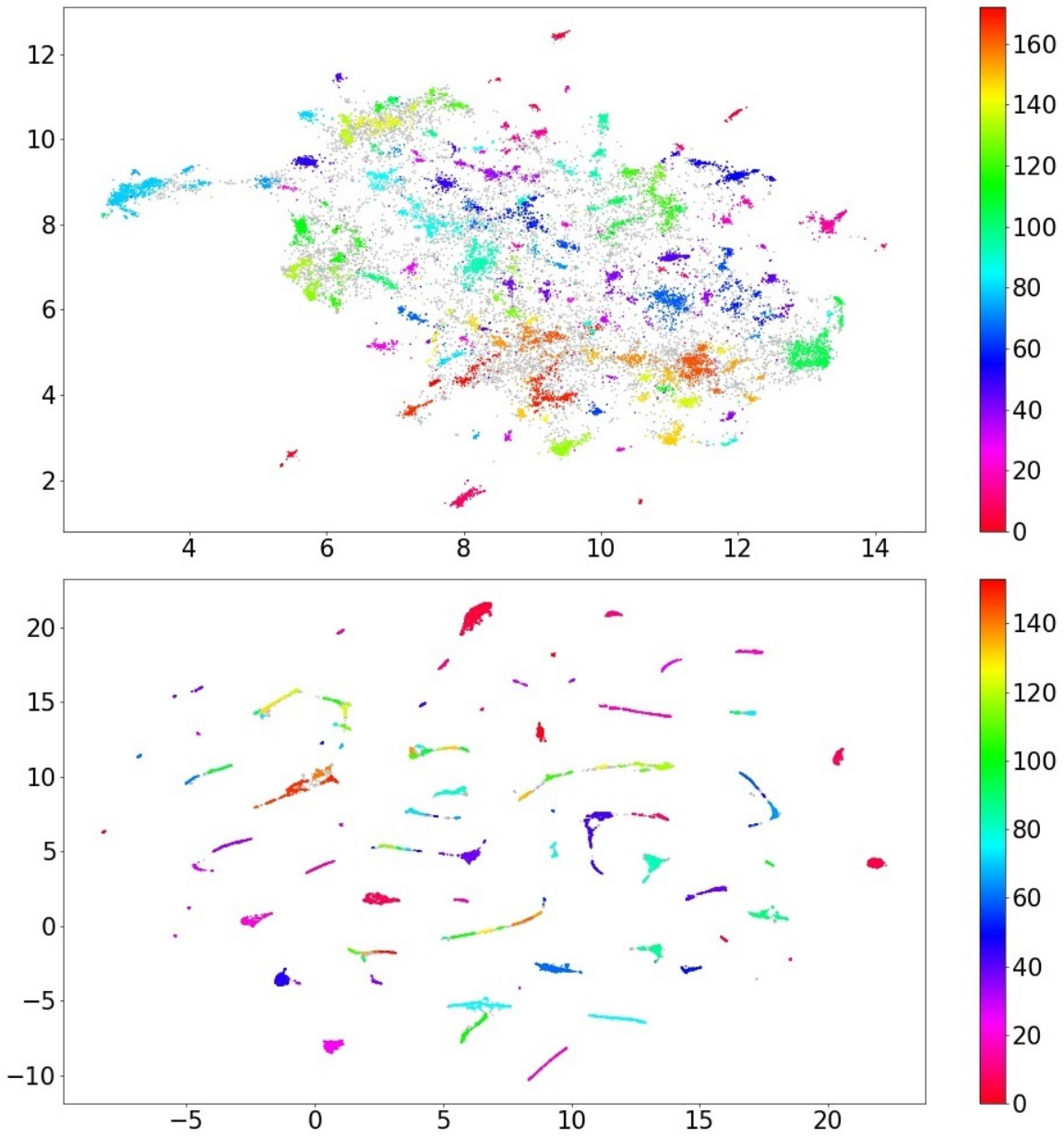

Figure 3.6: Visualization of API question sentence embedding before and after contrastive training

embedding visualization of the training samples after contrastive training. The figure reveals clear cluster patterns among the query embedding vectors, with most APIs forming dense clusters and relatively distinct margins between clusters. This visualization supports the hypothesis of contrastive training, indicating that semantic-equivalent queries are pulled together while irrelevant vectors are pushed apart.

3.7.2 Application of CLEAR on the Real-world Practice

CLEAR, BIKER, and RACK were evaluated on 50 recent Java-related questions from Stack Overflow. Among the top 10 recommended APIs, CLEAR successfully provides recommendations for 34 queries, for 23 queries, and RACK for 4 queries. For demonstration, two random examples solvable exclusively by CLEAR were selected.

Table 3.6 presents the recommendation results of CLEAR for two recent Stack Overflow posts. The first post, dated January 8, 2021 at 18:13, demonstrates CLEAR 's capability to interpret time representations in multiple formats and correctly identify the keyword "convert". These results indicate that CLEAR avoids the lexical similarity pitfall concerning time formats and successfully recommends appropriate APIs. The second example, posted on January 15, 2021 at 17:25, involves a question about loading property files. Here, CLEAR accurately extracts the most relevant keywords, such as "property file" and "retrieve". Additionally, the synonym "reading" for retrieve is correctly recognized, further validating the approach. The above two case studies show that CLEAR is more effective for API recommendation in real-world applications.

3.8 Related Work

There are many existing studies on API recommendation, including API invocation sequences mining [166], dependency graph-based API phrases mining [12], API recommendation for feature requests [25], query-API keyword mapping with crowed knowledge [8], code snippet synthesis [43], similarity-based API recommendation with language model [21], and ensemble model using the previously existing technique [140].

McMillan et al. [166] first presented *portfolio*, an API recommendation tool that returns code snippets for a programming query. Thung et al. [25] introduced historical feature requests combining with official API documents information for API recommendation for new feature requests. Nguyen et al. [35] proposed GRALAN, a graph-based language model for object-oriented source codes. Liu et al. [73] improved the ranking of the top-10 result of GRALAN by introducing API usage path information to the graph system. Nguyen et al. [24] used statistical learning on the commit changes information for API recommendation. Gu et al. [7] first introduced a deep learning model to API learning which achieves end-to-end API sequence generation. CLEAR uses Distil-RoBERTa as the base model, which is different from DeepAPI. Rahman et al. [8] presented RACK, an API recommendation tool leveraging the real API usage data from Stack Overflow [141]. The difference between the RACK and CLEAR is that CLEAR uses a language model instead of keyword mapping. Huang et al. [21] proposed BIKER, which filters the candidate APIs based on the similarity against SO questions and then re-rank the candidates based on the similarity against official API documentation descriptions. The difference between BIKER and CLEAR is that CLEAR uses contrastive training instead of unsupervised training in the model-building stage.

3.9 Threats to Validity

Internal Validity. relates to the errors in the implementation of CLEAR and the baselines. The code has been checked to ensure the questions in the testing dataset are not included in the question base. The replication packages of the baselines were reused to ensure their correctness.

External Validity is about the quality of the dataset. Although the dataset collected from SO is being filtered by heuristic rules, there is still noise in the dataset due to the openness of SO. It is hard to automate the cleaning process and also extremely time-consuming if do it manually. These noises may impact the performance of the CLEAR.

Construct Validity relates to the suitability of this thesis's evaluation metrics. MRR, MAP, *precision@k*, and *recall@k* were used to measure the performance of this thesis's approach and baselines, which are widely adopted evaluation measures for recommendation systems [21, 8].

3.10 Conclusion

This chapter proposed CLEAR, a new approach for API recommendation tasks. CLEAR uses the BERT-based model for embedding, which produces the embedding of the whole sentence of an API query while taking semantic-related sequential information into consideration. It uses contrastive training to better capture the semantic of the API queries regardless of the lexical information. This thesis's experiment results confirm the effectiveness of CLEAR for both method and class-level API recommendation. This thesis's case study with CLEAR on the latest SO posts further demonstrates the practical value of CLEAR for API search. In the future, the plan is to extend CLEAR to other tasks like third-party API recommendation, Linux command search, code snippet search, and program patch search.

Chapter 4

Understanding and Mitigating the Uniqueness of Machine Learning API Recommendation

4.1 Introduction

Application Programming Interfaces (APIs) are built-in functions in software libraries that help developers build software more effectively. As machine learning recently made great progress in both theory and application, there is increasing interest in developing machine learning applications. While there are many publicly available open-source machine-learning libraries and APIs, searching the right APIs for specific tasks is not easy, especially for the plenty of green hands in the field of machine learning [167].

Although many API recommendation approaches that can help retrieve APIs with high accuracy have been proposed and extensively studied [21, 8, 7, 10], most existing approaches have been evaluated mainly on general programming tasks using programming languages such as Java. Little is known about their practical effectiveness and usefulness for ML programming tasks with dynamically typed programming languages such as Python. Apart from the different nature of programming languages, machine learning application development has different paradigms compared to traditional application development (relatively more deterministic and less statistically orientated) [167, 168], which is statistically orientated and requires many algorithms, mathematical operations, and data operations [169, 170, 171].

This thesis investigated the effectiveness of existing API recommendation approaches for Python-based ML programming tasks, which is of great value considering the increasing popularity of machine learning practices and the overwhelming number of machine learning questions on question answering websites such as Stack Overflow (SO). For example, there are more than 11K questions on Stack Overflow about API recommendation for TensorFlow

(an open-source library for machine learning) with about 300 new questions emerging each week[12]. this thesis focuses on the following research questions:

RQ1: What is the performance of existing API recommendation approaches on Python-based ML programming tasks?

To answer this question, this thesis presents an empirical study to explore the performance of existing state-of-the-art API recommendation approaches, i.e., BIKER [21] and DeepAPI [7], on Python-based ML programming tasks.

The evaluation process begins with the collection of 80,000 Python-based machine learning programming questions related to six popular ML libraries from Stack Overflow. For fair comparison, both BIKER [21] and DeepAPI [7] are retrained using this Python-based ML question dataset. The models' performance is then evaluated on 1,000 randomly selected ML questions that were excluded from the training data. This thesis's experiment results show that there exists a significant performance decline of these approaches on Python-based ML questions. An in-depth analysis was performed to explore the reasons. This thesis's analysis reveals that there exist two major reasons. First, most traditional Java programming tasks require only one API to solve, while ML questions often require many more APIs because ML development tasks often require customization of data processing, feature engineering, model architecture, optimization function, and hyperparameters, etc. Specifically, the average length of the API sequence in the answers to Java programming tasks is 1.42, while the average length is 5.50 for ML tasks. Second, most existing approaches only focus on increasing the hit rate of the first correct API recommended while ignoring the completeness of the answers recommended. Moreover, the multiple APIs of Python-based ML tasks pose greater challenges to the recommendation tasks for Python-based ML tasks. One notable observation is that there are several other reasons for the performance decline, which are closely related to the nature of machine learning (details are in Section 4.7.1).

In addition, there exist common library-specific API usages that can be useful for further improving the API recommendation for Python-based ML programming tasks. For example, when constructing a machine learning model in Keras, *Keras.model.Sequential* has to be used as a container for model layers such as *keras.layers.Conv2D, keras.layers.Dense*. Current API recommendation approaches cannot capture the above information about API usage as they do not consider the relationship between API calls when recommending APIs for a programming task.

RQ2: Can the performance of the existing API recommendation approaches on Python-based ML tasks be improved?

Based on the findings from RQ1, this thesis propose a simple but effective booster, i.e., FIMAX , which can significantly improve the performance of existing API recommendation approaches for Python-based ML programming tasks by leveraging API usage patterns of ML libraries.

[12]https://stackoverflow.com/questions/tagged/tensorflow

Specifically, the study employs frequent itemset mining [172] to identify recurring API usage patterns within call sequences extracted from Stack Overflow answers to ML programming questions. The recommendation results of the existing approaches are extended with the API usage patterns. The evaluation shows that the proposed approach improves the existing state-of-the-art API recommendation approaches by up to 54.3% and 57.4% in MRR and MAP, respectively. A user study has also been employed in which 14 developers were divided into two groups using different tools to answer 20 ML questions randomly sampled from the testing dataset. On average, the group using the FIMAX can improve answer correctness by 36% and save answering time by 40%.

This thesis makes the following contributions:

- This thesis performs the first empirical analysis of existing API recommendation approaches on Python-based ML programming tasks, revealed their performance degradation on Python-based ML programming tasks, and summarized the reasons for the degradation.
- This thesis proposed a simple but effective approach, i.e., FIMAX , to augment the API sequences retrieved by existing approaches to boost their performance of API recommendations.
- Both the quantitative evaluation and user study show that FIMAX can help developers find the correct APIs for Python-based ML programming tasks more efficiently and accurately, compared with state-of-the-art baselines.
- This thesis released the source code of this thesis's tool and the dataset of the experiments to help other researchers replicate and extend this thesis[13].

4.2 Empirical Study

4.3 Empirical Study Setup

This section describes the research questions of this thesis, the data collection approach, and the analysis methodology.

4.3.1 Research Questions

This thesis is organized by the following two research questions:

RQ1: What is the performance of existing API recommendation approaches on Python-based ML programming tasks?

[13]https://doi.org/10.5281/zenodo.6360250

This research question examines whether state-of-the-art API recommendation approaches (BIKER and DeepAPI) demonstrate strong performance when applied to Python-based machine learning questions.

RQ2: Can we improve the performance of the existing API recommendation approaches on Python-based ML programming tasks?

This research question seeks to identify potential enhancements for existing API recommendation approaches, building upon the findings from RQ1.

4.3.2 Subjects of Study

To investigate the performance of existing API recommendation approaches on Python-based ML questions, six popular Python ML libraries were selected according to the number of downloads in the Python Package Index (PyPI) [173], which is the Python library management program.

This thesis manually identifies the machine learning-related packages from the top PyPI package list[14], which is ranked by number of downloads. According to PyPI statistics [174] (as of Dec. 2021), the six libraries that rank by monthly downloads are NumPy, Pandas, Scikit-Learn, PySpark, TensorFlow, and Keras. NumPy [175] is a fundamental library used in Python ML development primarily for numeric array operations. Pandas [176] is a library for data analysis of tabular data such as data tables. It provides a tabular view of parallel data and also APIs covering basic table operations such as filtering, sorting, and indexing. Scikit-learn [177] is an important library in statistical machine learning library that provides machine learning and scientific algorithms such as matrix operations, algebraic equations, and differential equations [178]. TensorFlow [179] and Keras [180] are two popular deep learning libraries. TensorFlow, developed by Google, is one of the most commonly used machine learning libraries in the industry as it supports many industry-friendly features such as distributed training and a visual debugging environment [167]. Keras offers a user-friendly stack-style API for building and training deep learning models. It encapsulates the detailed execution process in high-level APIs. PySpark [181] is a Python adapter of Apache Spark. It allows developers to use Spark application features such as Spark SQL, Distributed Data Frame, etc. in Python.

The libraries studied cover most of the current industrial practice of machine learning and also represent the key aspects of machine learning developments. In addition, they cover the area of data preprocessing and data object processing (NumPy and Pandas), scientific computing (SciPy and Scikit-learn), distributed machine learning (PySpark), and deep learning (TensorFlow and Keras). Although NumPy and Pandas are not libraries for building neural network structures, they are integral to machine learning development because they are widely used as the basis for data object computation in machine learning model development. NumPy APIs are typically utilized in the middle of the TensorFlow modeling

[14]https://hugovk.github.io/top-pypi-packages/

Table 4.1: The details of studied machine learning libraries in this work

Library	Description	#APIs in documents	# Unique APIs in dataset	#SO questions
Numpy	A library for multi-dimensional arrays and matrices operations.	1,373	1,151	29,771
Pandas	A library for data manipulation and analysis.	1,367	913	16,197
Scikit-learn	A library for machine learning algorithms.	1,154	1,082	6,376
Keras	An interface for artificial neural networks.	1,407	1,233	8,728
TensorFlow	A library for deep learning developed by Google.	8,028	1,591	10,263
PySpark	An Interface for Apache Spark in Python.	860	785	8,861

API sequences. Removing the NumPy APIs would leave the API sequences incomplete. Thus, Numpy and Pandas are also included in this work.

Table 4.1 shows the details of the six libraries studied. The number of APIs is calculated by counting the number of documented function nodes in the latest version of the packages using a Python AST parser. After removing duplicates for each package, the number of unique APIs in the experiment dataset is obtained. The number of SO posters for each package is collected from the Python ML dataset, as shown in Section 4.3.3.

4.3.3 SO Post Collection

The Python-based ML question dataset was constructed following established methodology [21] to extract programming questions and their accepted answers from Stack Overflow using the Stack Exchange data explorer [182]. Questions were selected based on six library names used as Stack Overflow tags, retaining only those with accepted answers.

API extraction from accepted answers employed heuristic methods. A specialized heuristic parser was developed to identify method names from code snippets, with package and class names inferred from import statements. For questions containing multiple APIs, all relevant APIs were concatenated into a comprehensive list. To ensure data quality, only answers containing at least one API were retained, with the additional constraint that all APIs must belong to the same library. This rigorous selection process yielded a final collection of 80,196 high-quality question-API pairs across the target libraries.

4.3.4 Evaluation Metrics

The API recommendation approaches were evaluated using Mean Reciprocal Rank (MRR) and Mean Average Rank (MAP), which are two commonly used evaluation metrics in information retrieval and recommendation system evaluation [183, 184, 185, 186, 187, 8, 188, 21]. Both MRR and MAP range from 0 to 1. MRR describes the rank of the first correctly recommended API in the recommendation list. It is calculated by the inverse rank of the first match [21]. A high MRR indicates that the rank of the first correct match is high. MAP checks the ranks of all correct matches instead of focusing only on the first correct answer. A high MAP indicates that there are multiple correct matches in the recommendation and thus the recommendation is complete.

Table 4.2: Performance of BIKER and DeepAPI on Python-based ML questions

Dataset	Approach	Evaluation Metrics	
		MRR	MAP
Python ML	BIKER	0.271	0.115
	DeepAPI	0.176	0.068
Java JDK	BIKER	0.554	0.505
	DeepAPI	0.188	0.153

4.4 RQ1: Performance Comparison

To investigate RQ1, the thesis trains and evaluates BIKER and DeepAPI on the Python-based ML questions collected from Stack Overflow (see Section 4.3.3 for details).

In addition, an in-depth analysis was conducted to investigate possible reasons for the difference in performance of the two approaches on traditional programming tasks and Python-based ML programming tasks.

4.4.1 Experimental Setup

Training: To adapt BIKER and DeepAPI for Python-based ML questions, the original Java question data was replaced with Python-based ML question data for model retraining. As BIKER utilizes both question titles and API documentation for recommendations, the Java documentation was similarly replaced with Python documentation.

Python documentation was collected using a Python AST parser to extract docstrings from each target library. The parser systematically traversed from the root directory, capturing all docstrings within both "FunctionDef" and "ClassDef" nodes.

For DeepAPI evaluation, the model was trained and tuned from scratch on the Python-based ML question dataset. The training configuration matched the original paper's specifications [7], implemented on a single NVIDIA V100 GPU with 32GB memory.

Testing: To evaluate the performance of the examined two API recommendation models, this thesis randomly selects 1k samples for each library for building the test datasets. According to previous studies on Stack Overflow [189, 190, 191], Stack Overflow contains not only API recommendation questions, but also other types of questions, such as comparing different implementations, asking for an explanation, and asking for program debugging etc. Therefore, to reduce noise in the test data, this thesis manually examines each question to remove the questions that are not suitable for API recommendations. The questions removed in the manual analysis either do not ask for API recommendations or provide information that is too general to derive an API recommendation from. For example, questions that ask for explanations, such as "*Why am I not seeing NumPy's Deprecation Warning?*", questions that ask for debugging help, such as "*MNIST ValueError when checking target in Keras*", and questions that are too general to recommend APIs, such as "*Constraints not working in Optimization using SciPy*". The authors of this work independently go through each

question and identify the API recommendation-related questions. The agreement among the researchers measured by Cohen's Kappa coefficient is 0.87, which is a relatively high-level of agreement. **Test Set Construction:** After removing irrelevant questions, the first 100 valid samples for each library were selected from the remaining pool to constitute the final test set. To ensure fair evaluation, all test data were rigorously excluded from the training dataset.

4.4.2 Performance Difference

Table 4.2 shows the MRR and MAP of BIKER and DeepAPI on the Python ML question dataset. In addition, the performance of BIKER and DeepAPI on the Java JDK question dataset used in their original papers is also included [21, 7]. In general, the performance of both approaches significantly (p-value <0.01) decrease when applied to the Python ML dataset compared to their performance on the Java JDK dataset.

For BIKER, the MRR declines from 0.554 on the Java JDK dataset to 0.271 on the Python ML dataset (a 51.0% decrease), and the MAP declines from 0.505 to 0.115 (a 77.2% decrease). DeepAPI exhibits a similar trend for both MRR and MAP, with MRR declining by 6.0% and MAP declining by 55.0%. On the Java JDK dataset, the MRR and MAP of both approaches are at a similar level, whereas on the Python ML dataset, the MAP for both approaches is approximately half of their respective MRR values. Since MAP measures the ranks of all correct matches instead of focusing only on the first correct answer like MRR, the above results show that the existing API recommendation approaches have challenges with the API recommendation completeness, i.e., they can hardly capture the entire set of correct answers.

4.4.3 Analysis of Performance Decline

Motivated by the significant performance difference shown in Section 4.4.2, this thesis further investigated the two datasets (i.e., Java JDK question dataset and Python-based ML question dataset) and the recommended APIs of both BIKER and DeepAPI. This thesis found two main reasons contributing to the performance decline of BIKER and DeepAPI on the Python-based ML question dataset, i.e., compared to the Java question data, answers of the Python ML questions often require more APIs (see Section 4.4.3 for details) and the existing API recommendation approaches mainly focus on increasing the hit rate of the first correct API recommended while ignoring the completeness of the answers recommended and do not consider the common API usages (see Section 4.4.3 for details). In addition to these two major reasons, minor factors may also contribute to the performance decline of existing API recommendation approaches. Further details are provided in Section 4.7.1.

Python ML Questions Require More APIs

This thesis finds that the average length of API call sequence in the answers of the Python-based ML questions is longer than that of the answers for Java SDK questions, which makes the recommendation for Python-based ML questions more challenging. Specifically,

the average length of the API call sequence in the ground-truth answers of the Java JDK questions is 1.42, while the average length of the API call sequence in the answers for Python-based ML questions is 5.50 In other words, one API can solve a Java JDK question, while five APIs are required to solve a Python-based ML question on average. The median API call sequence length also supports this observation, i.e., one in Java versus five in Python ML.

Due to the fact that most answers to Java JDK questions only contain a single API, existing API recommendation approaches such as BIKER and DeepAPI mainly focus on increasing the probability of the first correct answer (i.e., MRR), while ignoring the completeness of the recommended APIs (i.e., MAP). Specifically, given a question, BIKER first obtains candidate APIs from a list of similar questions, and then re-rank the collected APIs according to the similarity between the question title and the documentation of each API to ensure the correctness of the first answer without considering other APIs that have low similarity scores to the question title if even they are essential for answering the question. This hurts the completeness of the recommendation. For DeepAPI, it uses an LSTM encoder-decoder sequential model designed for the sequential task. This mitigates the problem BIKER faces yet still suffers from low performance. Considering that Python-based ML questions usually contain 5 or more APIs, the API recommendation in this scenario should consider both MRR and MAP, i.e., the first correct answer and all correct answers are ranked higher.

Existing Approaches Did not Consider Common APIs Usages

Table 4.3 presents a Python-based ML question, its accepted answer, and the corresponding API recommendations generated by BIKER. The question involves converting a 4096-dimensional output shape to 2048 dimensions. The table includes the accepted code solution, which contains a sequence of six relevant APIs. For clarity, the APIs successfully identified by BIKER are displayed in bold font, while those not recognized by the approach appear underlined in the code snippet.

Specifically, it first loads the VGG16 model except for the last layer and freezes the model layers. Then, it adds a new dense layer to the model with the required feature size. BIKER can correctly suggest "*keras.add*" and "*keras.VGG16*", but miss the prerequisite API, "*keras.Sequential*". In Keras, "*keras.Sequential*" acts as a container for all model layers, which means that the layer operation APIs must be called after a model container is initialized with "*keras.Sequential*". BIKER fails to recognize this API usage pattern. DeepAPI's recommendation results also show similar cases.

The above example shows that existing API recommendation approaches often neglect the relationships between the APIs involved when recommending APIs for a given question. Motivated by this, this thesis further investigates the co-occurrence of the involved APIs in the experimental dataset. The results are in Table 4.4, from which the API "*Keras.add*" occurs 1947 times, and "*Keras.Sequential*" co-occurs 1188 times together with "*Keras.add*" out of the 1947 times (i.e., 61%). This reveals the frequent usage patterns of APIs in solving

Table 4.3: An example of Python-based ML question (Bold APIs are APIs that are matched by BIKER, underlined APIs are APIs that are missed by BIKER).

Question: convert vgg16 shape output from 4096 features to 2048

Accepted code snippet to answer this question:

```
vgg16_model = keras.applications.vgg16.VGG16()
model = Sequential()
for layer in vgg16_model.layers[:-1]:
    model.add(layer)
model.layers.pop()
# Freeze the layers
for layer in model.layers:
    layer.trainable = False
# Add 'softmax' instead of earlier 'prediction' layer.
model.add(Dense(2048, activation='softmax'))
# Check the summary, and yes new layer has been added.
model.summary()
```

Ground-truth APIs:
keras.VGG16, keras.Sequential, **keras.add**, keras.pop, keras.Dense, keras.summary

Recommended APIs by BIKER:
keras.reshape, keras.concatenate, keras.shape, **keras.add**, keras.Input, keras.ones, keras.pad_sequences, **keras.VGG16**, keras.transpose, keras.arange

Table 4.4: Occurrence of API usage patterns

Base	Add	co-occurrence	confidence
keras.add	keras.Dense	972	0.7086
keras.add	keras.Sequential	1,188	0.6103

a particular task, i.e., two or more APIs co-occur frequently. Also, there exist numerous pairs of APIs that are highly likely to co-occur, which can be considered as API usage patterns.

The above analysis motivates us to extend the API recommendation result by existing tools (e.g., BIKER and DeepAPI) by including API co-occurrence relationships to improve the performance of API recommendation. Specifically, when an API is recommended, this thesis's approach can identify potential API usage patterns related to it and further append the involved APIs from the API usage patterns into the recommended API list. Such an extension could improve current API recommendation approaches by increasing the completeness of the APIs recommended.

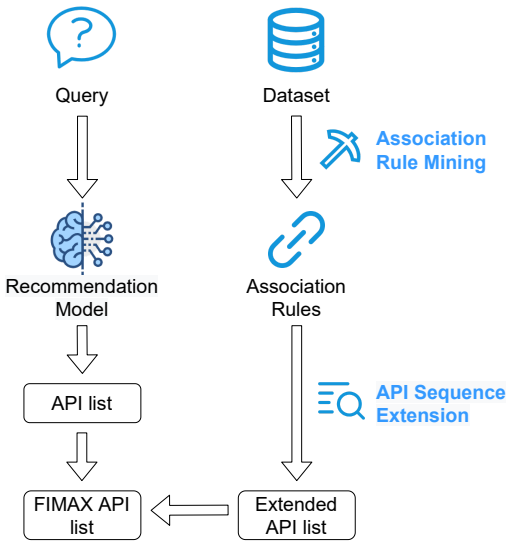

Figure 4.1: Overview of FIMAX

4.5 RQ2: Enhancement Solution

RQ1 has demonstrated the significant performance degradation of existing API recommendation approaches on Python-based ML questions. Inspired by the findings of RQ1, a frequent itemset mining-based approach was proposed, named FIMAX , that aims to improve the performance of existing API recommendation approaches for Python-based ML questions by extending existing API recommendation results with API usage information, i.e., co-occurrence relationships of APIs.

4.5.1

Figure 4.1 shows the workflow of FIMAX . First, FIMAX generates the co-occurrence relation-based API usage patterns of APIs with an association rule mining technique on API call sequences from the answers of ML programming tasks. Then, FIMAX further extends the recommendation from an API recommendation model with the mined the API usage patterns.

Association Rule Mining.

FIMAX first applies the Apriori algorithm [192] to create a set of API association rules (i.e., API usage patterns) from the Python-based ML question dataset. Specifically, this thesis collect the API sequence from the answer of each SO post in the Python-based ML question dataset, which serves as input to the Apriori algorithm for rule mining. The Apriori algorithm generates a pattern hypothesis by randomly selecting two or more APIs as the base itemset A and then randomly selecting one or more APIs as the extension itemset B.

The algorithm calculates the confidence and support of the pattern hypothesis $\{A \Rightarrow B\}$ and compares it to the confidence and support threshold specified by the user. The Apriori algorithm accepts an association rule if the *confidence* and *support* of the rule are higher than the threshold values specified [192].

Specifically, *Support* indicates the frequency of an API association rule with respect to the entire dataset. *Confidence* indicates the percentage of one or more extended API(s) (e.g., item set B) found to be true given a base rule (e.g., item set A).

Let A, B be the antecedent and the consequent mined from a list of API T, the *support* of $A \Rightarrow B$ over T is

$$support(A \Rightarrow B) = \frac{freq.(A, B)}{|T|}$$

and the *confidence* of $A \Rightarrow B$ is

$$confidence(A \Rightarrow B) = \frac{freq.(A, B)}{freq.(A)}$$

As an example, the Apriori algorithm proposes an API usage pattern {*"tensorflow.session"*$\Longrightarrow$*"tensorflow.run"*}. Then the algorithm calculates the *confidence* and *support* for this pattern. The *confidence* can be interpreted as the percentage of occurrences of *"tensorflow.run"* given *"tensorflow.session"*, and the *support* of {*"tensorflow.session"*$\Longrightarrow$*"tensorflow.run"*} is the percentage of the co-occurrences of *"tensorflow.session"* and *"tensorflow.run"* over the size of the entire corpus. If both *confidence* and *support* are greater than the specified thresholds, the proposed rule{*"tensorflow.session"*$\Longrightarrow$*"tensorflow.run"*} will be added to the table of item set patterns along with the *confidence* and *support* values.

API Sequence Extension.

Given a list of recommended APIs R, generated by an API recommendation approach, e.g., BIKER, FIMAX searches for the extension rules for each API in the list. If there are rules that match a particular API, FIMAX appends the associated APIs to R according to the association rules created in Section 4.5.1 for that API. If there are multiple rules that are eligible for extension, only the rule with the highest support score will be utilized. The final API recommendation list consists of the top K original APIs concatenated with the extended APIs. The extended APIs are ordered by confidence score.

Table 4.5 shows an example of the extension. Following the extension algorithm, FIMAX uses Rule B and Rule C for extension. In case there are duplicate APIs after the extension. For a repeated recommendation, the first occurrence was kept, and the duplicates were removed.

Table 4.5: An example of API extension of FIMAX

APIs recommended:
numpy.range, numpy.reshape, numpy.arange

Rule A:
$\{numpy.range \Rightarrow numpy.empty, support = 0.0103\}$
Rule B:
$\{numpy.range \Rightarrow numpy.append, support = 0.0168\}$
Rule C:
$\{numpy.reshape, numpy.arange \Rightarrow$
$numpy.array, support = 0.02204\}$

Extended API list:
numpy.range, numpy.reshape, **numpy.append**, **numpy.array**, numpy.arange

Table 4.6: Performance of FIMAX on BIKER and DeepAPI.

Library	BIKER		BIKER+FIMAX		Improvement		DeepAPI		DeepAPI+FIMAX		Improvement	
	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP
Numpy	0.43	0.16	0.56	0.21	30%	27%	0.21	0.06	0.49	0.16	127%	142%
Pandas	0.11	0.05	0.19	0.06	67%	21%	0.22	0.07	0.27	0.09	20%	23%
Scikit-learn	0.02	0.01	0.04	0.04	100%	97%	0.17	0.06	0.18	0.06	7%	-8%
Keras	0.04	0.01	0.06	0.02	7%	89%	0.21	0.07	0.26	0.10	25%	29%
Tensorflow	0.25	0.06	0.35	0.11	37%	61%	0.29	0.10	0.40	0.15	37%	46%
Pyspark	0.20	0.10	0.40	0.10	71%	126%	0.30	0.10	0.50	0.10	82%	68%
Overall	0.18	0.06	0.27	0.08	48%	53%	0.23	0.08	0.34	0.12	54%	57%

4.5.2 Experiment Setup

As discussed in Section 4.5.1, FIMAX employs the Apriori algorithm for association rule mining, which relies on two key parameters—*support* and *confidence*—that significantly influence its output. To determine optimal parameter values, the experiments explore support thresholds ranging from 0.002 to 0.01 in increments of 0.001, alongside confidence thresholds from 0.1 to 0.9 in steps of 0.1. Please note that both BIKER and DeepAPI can recommend many APIs for a given question, e.g., BIKER recommends up to 50 APIs, as a result, there will be a large number of candidate rules if all the recommended APIs are considered, while only a few API candidates are likely to be correct in practice. Extending the wrong API would degrade the performance of a recommendation approach. To control the recommendation scope, the number of APIs to be extended is limited to a threshold value K. To determine the optimal parameters, a grid search is conducted over all combinations of the support threshold, confidence threshold, and top K values, with each combination's MRR value computed for performance comparison.

The result shows that the BIKER+FIMAX model performs best when the confidence threshold is equal to 0.1, the support threshold is equal to 0.02 and top_k is equal to 6, while the DeepAPI+FIMAX model performs best when the confidence threshold is equal to 0.1, the support threshold is equal to 0.02, and top_k is equal to 10. The reason for the different best parameter settings for BIKER and DeepAPI can be these two approaches adopt different mechanisms in API recommendation and generate different recommended APIs. It also shows the necessity of parameter tuning when applying FIMAX to a new API recommendation approach. The model training and API recommendations for DeepAPI are performed on an Nvidia V100 GPU, while BIKER and FIMAX utilize an Intel i7-4790 CPU for API recommendation tasks.

4.6 Evaluation

4.6.1 Performance Analysis of FIMAX

The evaluation assesses FIMAX 's effectiveness in enhancing two API recommendation approaches, which are BIKER and DeepAPI, on Python-based ML questions by comparing their original performance against their FIMAX -augmented versions.

Table 4.6 shows the evaluation results for BIKER, DeepAPI, BIKER+ FIMAX , and DeepAPI+FIMAX on questions from each Python ML library. The table shows that the FIMAX significantly increased the performance of both models. Overall, the MRR and MAP of BIKER increased by 48.36% and 53.46%, respectively, after applying the FIMAX . The MRR and MAP of DeepAPI increase by 54.28% and 57.36%, respectively, after applying FIMAX . Moreover, the performance increase of MAP is larger than MRR for both models, suggesting that FIMAX has a positive effect on the completeness of API recommendations. The Wilcoxon signed-rank test ($p<0.05$) also suggests that BIKER and DeepAPI can achieve significantly better performance after applying FIMAX .

In addition, BIKER+FIMAX has a noticeable improvement over the performance of BIKER for each library. The MAP of BIKER in the PySpark package shows the most significant improvement, i.e., a 126.46% increase after applying FIMAX .

The main reason for this improvement might be that PySpark is an analytics engine for Big Data, and most of its API sequence follows the Map-Reduce related patterns that can be learned by FIMAX .

For DeepAPI, the MRR for the NumPy library increased by 127.6% and the MAP increased by 142.7% after the FIMAX was applied, which is the most significant improvement of DeepAPI after applying FIMAX . One of the possible reasons for this improvement is that NumPy is designed for manipulating arrays and matrices exclusively. Thus, its API usage patterns can be much more targeted than other libraries, which makes them easy for FIMAX to learn.

Note that although FIMAX can significantly improve BIKER and DeepAPI on most Python ML libraries regarding both MRR and MAP, a performance decline can be observed on the Scikit-learn library of DeepAPI after applying FIMAX . Specifically, the MAP of

Table 4.7: Time cost of BIKER, DeepAPI, and FIMAX .

Approach	Device	Training	Recommendation
BIKER	Intel i7-4790 CPU	N/A	5.4s/query
DeepAPI	Nvidia V100 GPU	14 hrs	0.57s/query
FIMAX	Intel i7-4790 CPU	3 mins	0.00576s/query

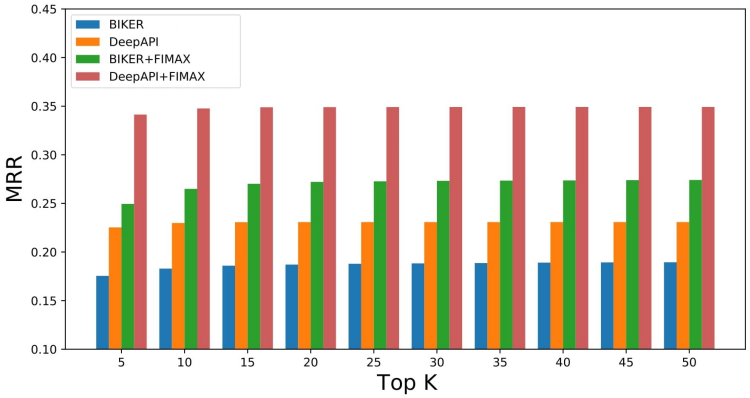

Figure 4.2: Performance of FIMAX under different top-k setting regarding MRR

DeepAPI+FIMAX declines -7.99% compared to the performance of DeepAPI. One of the possible reasons for this is that the Scikit-learn library contains many statistical algorithms that are rarely used and queried on SO, making it difficult for FIMAX to learn the correct API usage patterns while returning many false positives, which further causes a performance decline.

Table 4.7 shows the time cost of FIMAX and the baselines. In the training phase, BIKER requires no training, and DeepAPI requires 14 hours to train the model. In the recommendation phase, BIKER requires 5 seconds, while DeepAPI requires 0.5 seconds for each query. The time cost of FIMAX is only 3 minutes in training and 0.00576 seconds per query in the recommendation phase. Compared to the time cost of BIKER, the time cost of FIMAX is negligible in both the training and recommendation phases.

4.7 Discussion

4.7.1 More Reasons for Performance Decline

This thesis presented two major reasons that contribute to the decline in the performance of existing API recommendation approaches. In addition, there are several other reasons for the performance decline that were found in the investigation of the recommendation results, which are related to the nature of machine learning. This thesis presents these reasons as follows to

motivate the future directions to improve the API recommendation on Python-based ML questions.

Library Bias: Some users specify the library name in their questions and others do not. When a user does not specify the library name, the API recommendation approach suffers from popularity bias, i.e., the APIs of more popular libraries in the training dataset are more likely to be recommended. For example, the API used for a fully connected layer in TensorFlow is "*tf.contrib.layers.fully_ connected*" while it is "*keras.layers. Dense*" in Keras. When a user asks questions about implementing a fully connected dense layer without mentioning the library name, the model tends to recommends "*keras.layers.Dense*" rather than "*tf.contrib.layers.fully_ connected*". This is because there are many more questions about "*keras.layers.Dense*" than that of "*tf.contrib.layers.fully_ connected*" on Stack Overflow, and rarely used APIs are less likely to be recommended. Such bias can be mitigated by specifying the library name in the question or balancing the training examples. This also motivates the practical need for fairness study in API recommendations [193, 194].

Multiple solutions: There are questions that have multiple solutions. For example, a question asking how to iterate through a Pandas column can be answered with solutions such as "*df.iterrows()*" or "*df.iteritems()*". Although both are correct recommendations, following the existing evaluation method [21, 7], only the recommendation that matches the answer in the collected dataset is considered correct. Moreover, this finding also indicates how inflexible the existing evaluation method for API recommendations is and motivates the need for new evaluation criteria.

Documentation issue: Another possible reason is that the format and organization of the documentation of Python-based ML libraries are different from those of Java JDK, which mainly affects the performance of BIKER. BIKER employs the documentation for similarity-based API re-ranking under the assumption that the documentation can provide description of the corresponding API. However, the API documentation of the Python ML libraries contains a detailed description of the input, output, caveats, and examples. This severely disturbs the recommendation model when calculating the similarity between the documentation and the question, resulting in performance degradation. This motivates the need for documentation understanding and key information extraction to automatically reorganize various sections of Python-based ML API documentation before applying BIKER.

4.7.2 Performance Against Different Top-k Settings?

Both BIKER [21] and DeepAPI [7] generate ranked lists of top k APIs for a given query. Following their evaluation methodology, this study examines the top 10 recommended APIs in two scenarios: (1) the original API sequences produced by BIKER and DeepAPI, and (2) the extended sequences generated by BIKER+FIMAX and DeepAPI+FIMAX . However, since the number of APIs required to solve a Python-based ML question is much larger than that of a Java JDK question (details are in Section 4.4.2), this thesis further evaluate the performance of FIMAX under more recommended APIs, i.e., from 5 to 50 with 5 as the interval.

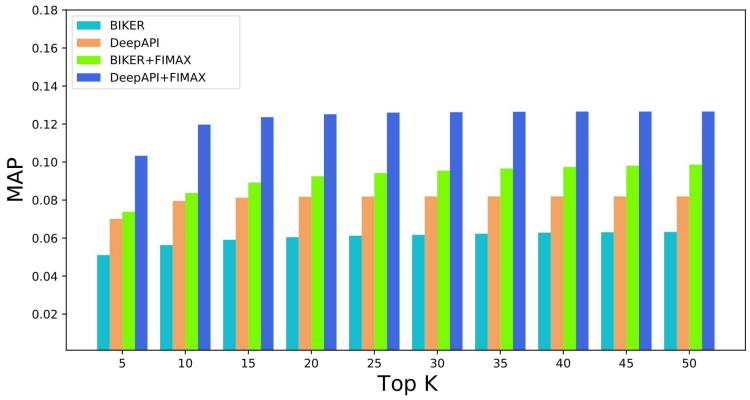

Figure 4.3: Performance of FIMAX under different top-k setting regarding MAP

Figure 4.2 shows the MRR of BIKER, DeepAPI, BIKER+FIMAX and DeepAPI+FIMAX under different top-k configurations. The MRR values of BIKER and DeepAPI are below 0.20 and 0.25 from the top 5 to the top 50 recommendation results, respectively. Also, the performance of BIKER and DeepAPI remains almost the same from the top 5 to the top 50 recommendations, which means that BIKER and DeepAPI cannot hit more correct APIs even if more recommendations are included.

After FIMAX is applied to both BIKER and DeepAPI, the performance of both approaches increases dramatically and remains stable across different top K settings. Figure 4.2 shows that the performance of BIKER+FIMAX is higher than BIKER at each top-K setting and increases slightly until the top-15 recommendation, meaning that BIKER+FIMAX is able to recommend the correct first match within 15 recommendation results. Figure 4.3 shows MAP values of BIKER, DeepAPI, BIKER+FIMAX , and DeepAPI+FIMAX under top k configurations from 5 to 50. Similar to the MRR result, the MAP values of BIKER and DeepAPI remain almost the same from the top 5 to the top 50. While the MAP values of BIKER+FIMAX and DeepAPI+FIMAX increase from top 5 to top 15, which shows that the FIMAX extension approach is able to increase the completeness of the original recommendation result of BIKER and DeepAPI in top k settings from 5 to 50. Also, BIKER+FIMAX and DeepAPI+FIMAX are able to recommend correct APIs not only for the first 10 but also for the first 15 recommendation results.

4.7.3 Generalizability of FIMAX

In this thesis, FIMAX has been shown to be effective in improving the performance of API recommendation on Python-based ML programming tasks whose answer contains a long API call sequence, while its performance on traditional Java SO questions that require few APIs is unknown (details are in section 4.4.3). This section evaluates FIMAX 's generalizability through its application to Java JDK questions, utilizing the existing JAVA JDK dataset from BIKER [21] for experimental validation. FIMAX mines the API usage rules on the

Table 4.8: Performance of FIMAX on different Java SO questions regarding the length of API sequence in the answers.

Java Questions	Improvement		
	MRR	**MAP**	**count**
Single Answer (=1)	3.2%	3.2%	100
Medium Answer ([2, 5])	0.8%	5.2%	100
Long Answer (>5)	0.4%	6.9%	100
Overall	1.0%	5.0%	300

training dataset and tunes its parameters (i.e., support and confidence) by following the same process used in section 4.5.2. To better analyze performance across different problem complexities, the test dataset is categorized into three groups based on API requirements, i.e., Single (requires one API), Medium (requires two to five APIs), and Long (requires more than five APIs). For the single category, 100 questions were randomly selected from the original test dataset of BIKER. Meanwhile, it was found that most questions in BIKER's test dataset are single-answer questions. Thus, for the Medium and Long categories, 100 samples were randomly selected from BIKER's training data and excluded during the training of BIKER.

Table 4.8 shows the improvement of FIMAX +BIKER regarding MRR and MAP for different types of JAVA questions compared to the performance of BIKER. The result shows that MAP increases by about 5% for the three categories and MRR improves by 3.2% for Single answer. The increase in MRR results from cases where none of the original API recommendations are correct, but the extended API hits the correct answer. The experiment result on Java JDK questions is consistent with that of Python-based ML questions, i.e., the FIMAX has a positive effect on the completeness of API recommendations. The Wilcoxon signed-rank test result ($p < 0.05$) further confirms that FIMAX +BIKER could achieve significantly better performance than BIKER.

4.7.4 Threats to Validity

Internal Threat: Since BIKER and DeepAPI are designed for Java questions, necessary adaptations were required for Python-based ML questions. The published source code of BIKER and DeepAPI was reused for the baseline method with the dataset. The implementation of FIMAX and the process of applying FIMAX to BIKER and DeepAPI were carefully reviewed to ensure that FIMAX functions as intended. Therefore, the threat to internal validity is low.

Construct Threat: This thesis follows the existing work [21] and adopts the same metrics (i.e., MRR and MAP) for performance evaluation. MRR and MAP are computed by reusing the algorithm in the source code of BIKER. However, the performance of FIMAX could be different if other metrics are used. In the future, the plan is to examine FIMAX with other metrics, e.g., Precision and Recall, which are widely used in information retrieval [162].

External Threat: This thesis collects the Python-based ML questions dataset from the official data explorer provided by Stack Exchange [182]. Although the same criteria as BIKER were followed to filter noisy data, the dataset may still contain some noise. To mitigate this threat, the test dataset was manually verified, as described in Section 4.4.1. The inter-rater agreement among the authors, measured using Cohen's Kappa coefficient, was 0.8731, indicating a high level of agreement.

4.8 Related Work

API Recommendation: There are many other approaches for API recommendation other than BIKER and DeepAPI [8, 43, 166, 12, 30]. Rahman et al. [8] proposed Rack for class-level Java API recommendation using a customized co-occurrence-based data-mining algorithm on top of the Lucien engine. BIKER reported better performance compared to RACK at the class level. Raghothaman et al. [43] proposed SWIM for synthesizing code snippets using API usage pattern mining techniques on GitHub code repositories. It trains a query-to-API model with the API call sequences from GitHub, and then synthesizes code snippets for a given query using the model. The difference between FIMAX and RACK and SWIM is that FIMAX applies the Apriori algorithm for mining API usage patterns, while RACK uses a customized technique and SWIM uses a mixture of several algorithms for code synthesis. McMillan et al. [166] proposed *Portfolio* for recommending related functions in C++ using data mining techniques in code archives. It applies the PageRank algorithm as well as the function call graph for association model building, and a customized similarity metric for relevant function ranking. Chan et al. [12] proposed a graph search approach that improves the performance of *Portfolio*.

They first build an API call graph using the text phrase in API-related questions, and then apply a customized shortest path indexing scheme for result ranking. Their approach significantly improves the performance of *Portfolio*.

Existing studies on API recommendations have primarily focused on Java programming questions. This work investigates two state-of-the-art approaches, BIKE and DeepAPI, for Python-based machine learning programming tasks.

He et al. [75] proposed PyART for real-time Python API recommendation.PyART is able to recommend both third-party library APIs and project-specific APIs. A direct comparison between FIMAX and PyART was not conducted because the two approaches utilize different information for API recommendations in different scenarios, making such a comparison potentially unfair. Specifically, PyART focuses on real-time code completion and makes API recommendations based on the context of the programming tasks. This thesis's approach is applied to a programming QA system by providing a search service that accepts a programming question and returns a list of APIs.

Mining API Usages: There are many studies on mining API usage [195, 196, 54, 197, 45, 198]. Treude et al. [195] proposed SISE, an API documentation augmentation technique that provides usage insights to developers. It applies a pattern-based approach with consideration of part-of-speech tags for feature extraction and a supervised machine learning

approach for extracting insights from SO posts. In a comparative study with eight software developers, SISE was found to contribute the most useful information to API documentation. Moreno et al. [196] have proposed MUSE for mining code examples. Given a particular method, it returns a list of related code examples generated by a combination of code clone detection and static slicing, and ranks the result based on a set of usage-based heuristics rules. Zhong et al. [54] proposed MAPO for recommending related code snippet using frequent API usage patterns. MAPO builds a recommendation model using a frequent sub-sequence mining algorithm with source code extracted from Google code archives. For a given API method, it recommends the associated API calls based on the usage patterns captured by the model. Petrosyan et al. [197] proposed an approach to discover tutorial sections to explain a particular API type. They create a supervised text classification model for classifying tutorial fragments based on linguistic and structural features. Nguyen et al. [45] proposed API2VEC, a model that learns the semantic relationship between APIs using word embedding techniques. They build an embedding model using the CBOW word2vec model with the extracted API sequence from Java and C# code snippets. They demonstrate the usefulness of API2VEC with 3 example applications, including an example of a newly discovered API mapping between Java and C# code. They go one step further and develop an automated API mapping discovery tool called API2API based on API2VEC for API migration tasks between Java and C#. Jiang et al. [198] have proposed an unsupervised approach called FRAPT for finding relevant tutorial fragments for a given API. The difference between FRAPT and this thesis's approach is that FRAPT uses PageRank and a topic model-based algorithm, while FIMAX uses the Apriori algorithm for association rule mining.

4.9 Conclusion

This section examines the effectiveness of state-of-the-art API recommendation approaches, namely BIKER and DeepAPI, for Python-based ML programming tasks. An empirical study was conducted using programming questions related to six widely used Python-based ML libraries. The results reveal two primary factors contributing to the performance decline of existing approaches: (1) Python-based ML tasks frequently involve lengthy API sequences, and (2) common API usage patterns in Python-based ML programming are not effectively handled by current methods. Inspired by the findings, this thesis proposed FIMAX , which enhances existing API recommendation approaches by using API usage information mined from SO questions. This thesis's evaluation shows that FIMAX can significantly boost existing state-of-the-art API recommendation approaches. This thesis's user study further demonstrates the practical value of FIMAX for API recommendations.

In the future, more investigation is planned towards the effectiveness of FIMAX on other API recommendation approaches and programming tasks in other domains and languages.

Chapter 5

Demystifying Machine Learning API Misuse

5.1 Introduction

Over the last decade, Deep Learning (DL) libraries have played an important role in various domains, such as image processing [199], autonomous driving [200], face recognition [201], drug discovery [202], and natural language processing [203]. These libraries facilitate the implementation of complex deep-learning algorithms and have revolutionized the field of computer science. However, developers often face challenges when using the DL APIs within these libraries. Due to the fact that the DL development paradigm differs greatly from traditional software development [104], knowledge in traditional software development may not be directly applicable to DL APIs. Developers who lack familiarity with DL APIs and relevant concepts may frequently encounter API misuses.

Existing studies on API misuse have primarily focused on Java APIs of traditional software [59, 204, 205, 206]. However, there has been limited investigation specifically into API misuse within Python DL APIs [207, 208, 209]. Islam et al. [208] conducted a comprehensive analysis of DL bugs by manually examining over 2,500 StackOverflow posts and GitHub commits related to five popular DL frameworks. Although they categorized API bugs, they did not delve into the specific details of these instances. Baker et al. [207] performed a manual analysis of 15 TensorFlow bugs, identifying 7 API misuse patterns based on a dataset from a pre-existing DL defect study [210, 211]. However, their studies' limited scale raises concerns about their representativeness. Wan et al. [209] conducted an API misuse study on cloud computing service APIs for four major cloud computing service providers across three application directions. Their focus was on performance problems and user experience of commercial APIs, leaving the characteristics of API misuse in lower-level building-block APIs like PyTorch and TensorFlow unexplored. While prior studies have provided basic insights into DL API misuses, a comprehensive study in this area remains absent, which would be highly valuable as it can provide developers with guidance regarding potential pitfalls when using APIs in the development of new applications. This study

presents the first comprehensive DL API misuse investigation of two major DL libraries, TensorFlow and PyTorch.

The research began by collecting 18,794 bug-fix commits from the top 200 most-starred projects using either PyTorch or TensorFlow. A set of heuristic filters was then applied to eliminate irrelevant commits (e.g., changes that do not involve API names, API conditions, or API parameters), resulting in a refined dataset of 4,224 commits for manual analysis. The details of the data collection are described in Section 3.3.

From the 4,224 commits, 891 API misuses were successfully identified through manual investigation and analysis. To better understand the characteristics of DL API misuse, the analysis was conducted from three perspectives: API misuse categories, root causes of API misuse, and symptoms of API misuse. In addition, the feasibility of utilizing a state-of-the-art API misuse detector [207] to identify DL API misuses was investigated. The analysis reveals a fundamental distinction between identifying DL API misuses and traditional software API misuses. Traditional API misuse detectors [212, 213, 214, 215] primarily rely on static program analysis, whereas DL API misuse mainly involves issues related to data and device usages, such as returning incorrect float types or accidentally running GPU tasks on CPU. These issues often manifest as performance differences or incorrect results, rather than triggering errors that can be easily detected through static analysis. Furthermore, identifying these types of DL API misuses heavily depends on human experience and a comprehensive understanding of the source code semantics. Consequently, existing detectors may not be suitable for effectively detecting DL API misuses, given the substantial contextual shift and the unique characteristics compared to traditional software API misuses.

In response to the limitations of existing API misuse detection for DL API misuses, LLMAPIDet is proposed, leveraging Large Language Models (LLMs) to detect and fix DL API misuses.

The study bases its approach on ChatGPT [216], which has demonstrated significant potential in software engineering tasks such as code generation [217], bug repair [218], and program understanding [219], among comparable LLMs [84].

The primary idea of LLMAPIDet is to identify API misuse by applying a set of predefined rules to API usage instances. The proposed approach initiates the process by constructing a corpus of natural language API misuse rules by querying ChatGPT with the collection of API misuse code examples. For each API usage code snippet, the approach first utilizes ChatGPT to generate its code explanation. Subsequently, it employs a retrieval-based few-shot learning technique to provide the most relevant API misuse rules to ChatGPT, enabling it to determine whether the code snippet involves an API misuse. Then, the instruction will be sent to ChatGPT to generate the patch.

LLMAPIDet was built by prompt-tuning a chain of ChatGPT prompts on 600 out of 891 confirmed API misuses and reserving the remaining 291 API misuses as the testing dataset. This thesis's evaluation shows that LLMAPIDet can successfully detect 48 API misuses. Since ChatGPT's training data includes source code from publicly available open-source projects, an additional 4,359 DL API usage instances were collected from the latest versions of 10 GitHub projects built with PyTorch and TensorFlow—distinct from the 200 projects

used in the empirical analysis—to mitigate potential data leakage issues [220, 221]. All the versions are released after July 1st, 2023. This thesis's evaluation on the 4,359 DL API usage instances shows that LLMAPIDet can identify 119 previously unknown API misuses and successfully fix 46 of them, whereas the state-of-the-art tool could only detect 5.

The contribution of this thesis is as the following:

- This thesis presents the first large-scale analysis to demystify and detect DL API misuses in PyTorch and TensorFlow.
- This thesis creates a benchmark DL API misuse dataset including 891 instances of DL API misuse. This thesis provides detailed taxonomies regarding the types, root causes, and symptoms of DL API misuses.
- Motivated by the ineffectiveness of state-of-the-art API misuse detection on DL API misuses, this thesis further presents a novel LLM-based API misuse detector, i.e., LLMAPIDet , to detect and repair DL API misuses. This thesis's evaluation shows that this thesis's tool can outperform the state-of-the-art API misuse detection.
- This thesis provides a set of practical guidelines to help machine learning development teams develop reliable programs by avoiding and detecting DL API misuses.
- This thesis releases the dataset and source code of the experiments to help other researchers replicate and extend this thesis[15].

The structure of the rest of the chapter is as follows: Section 3.3 describes the approach used for the empirical study. Section 5.3 presents the empirical study and result analysis. Section 5.6 presents the evaluation of state-of-the-art API misuse detection on DL API misuses. Section 5.4 details the proposed LLM-based DL API misuse detector.

5.2 Overview

5.2.1 Data Collection

Experiment Library and Project Selection

To collect experiment data, PyTorch and TensorFlow were selected as the target DL libraries due to their wide acceptance in both industry and academia. PyTorch [222], officially released in 2016 by the FAIR research lab, has gained immense popularity in the field of deep learning research due to its flexibility and ease of use. On the other hand, TensorFlow [223], developed by the Google Brain team in 2016, has found extensive utilization in industrial settings owing to its comprehensive ecosystem and broad coverage of use cases. Note that other DL libraries have been excluded, such as Caffe [224], MXNet [225], Theano [226], and CNTK [227], due to their comparatively lower popularity and maintenance activity. Machine learning libraries

[15]https://anonymous.4open.science/r/LLMAPIDet_replication-C157

like scikit-learn [177], XGBoost [228], and LightGBM [229] are also excluded due to the significant differences in the paradigm and workflow between deep learning libraries and traditional statistical-based machine learning libraries.

To conduct this thesis's study, the top 200 most-starred projects for the two studied DL libraries (i.e., PyTorch and TensorFlow) from GitHub were collected, and also the complete commit history of each project was pulled and saved locally. The selected projects span from the years 2015 to 2022, with an average project creation date of 2018. The number of forks ranges from 248 to 21,624, with an average of 2,708 forks per project. The size of the repositories varies significantly, ranging from 7 MB to 19 GB, with an average size of 198 MB. The complete list of project names can be found in the replication package.

Data Filtering

For each experimental project, the commit changes involving DL APIs were identified using the API lists provided in TensorFlow and PyTorch documentation to filter out irrelevant commits. Following the methodology of previous research [59], API misuse was observed to typically involve a small number of line edits. Therefore, changes were restricted to fewer than 10 lines, considering both additions and deletions.

It should be noted that certain API-related commits were excluded from the study, including custom API document updates, string updates, typos, and logic changes, as these categories represent normal development updates rather than API misuses. Following the data collection process, the methodology of an existing study [230] was adopted, employing GumTree [231], an AST diff checker, to collect commit diffs at the AST tree structure level.

The investigation also revealed the presence of code clones, where identical code changes appeared across multiple commits. As the study focused on identifying API misuse patterns rather than code clones, only one instance of each unique code change was retained, with duplicates being eliminated. Additionally, code changes lacking any methods were removed. After applying these filtering steps, a final set of 4,224 commits was obtained.

5.2.2 Manual Analysis

For the manual analysis, three of the authors, with an average of 6 years of experience in DL development, were involved. To better understand the dataset, an exploratory data analysis (EDA) was conducted using a variation of Grounded Theory [232]. Following the approach of prior work [59], 200 randomly sampled commits were manually labeled, documenting whether each instance represented an API misuse along with details about the misuse types, involved API elements, violations, symptoms, and root causes. The labeling process continued until each API misuse received a distinct classification. On average, each sample required approximately 6.3 minutes to label, including time spent reviewing commit messages, code changes, API documentation, and contextual information within the code commits.

The collected information was summarized to determine classification dimensions and class names based on notes from the EDA process. Through this analysis, three categorization

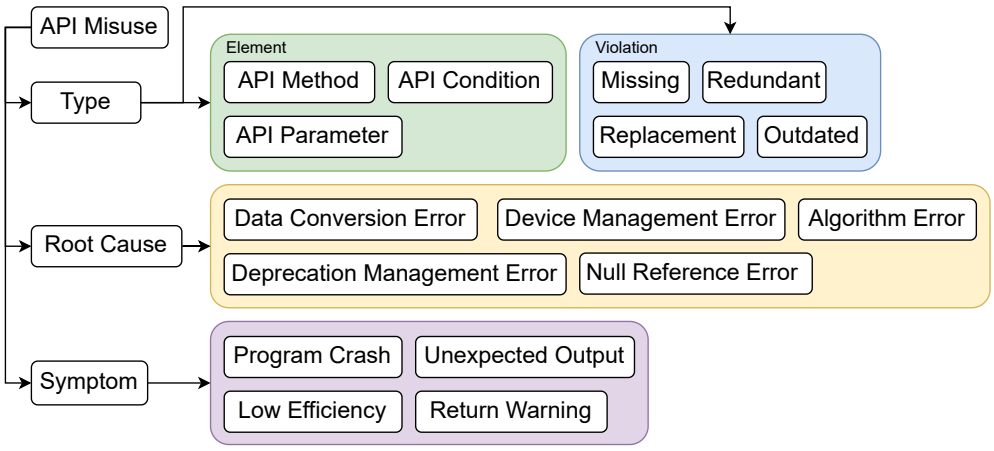

Figure 5.1: API Misuse Categorization

dimensions emerged: API misuse types, API misuse symptoms, and API misuse root causes. Figure 5.1 presents these dimensions and their corresponding classes.

Following the establishment of categories and classification rules, all 4,224 commits were systematically classified. Instances confirmed as API misuses were categorized according to the three identified dimensions. This process ultimately identified 891 API misuse instances, encompassing 311 unique API methods. These misuses cover approximately 10% of the APIs in PyTorch and TensorFlow libraries (3,622 APIs total).

API Misuse Types: In the analysis, API misuse types are categorized based on two sub-dimensions: API Violations and API Elements. Li et al. [230] categorized the type of violation into three types, i.e., *missing*, *redundant*, and *replacement*. In this thesis, the replacement class was extended to *Replacement* and '*Outdated* to reflect the significant difference in the characteristics of these two types in DL API misuse. The difference is that the *Replacement* API element represents the complete replacement from one API method to another, while the *Outdated* API element represents an outdated class or API method that requires a version update. Amann et al. [59] proposed a classification scheme comprising four elements of Java API misuse: *method call*, *condition*, *iteration*, and *exception handling*. In contrast, the current study identifies three types of API elements for Python DL API misuse based on empirical observations: *API method*, *API parameter*, and *API condition*. The iteration category was excluded from the classification as no such instances were observed. Additionally, the exception handling category was reclassified under root causes, as this alignment better reflects its conceptual relationship to API misuse patterns.

API Misuse Root Cause: To better understand the technical characteristics of API misuse, existing work was systematically categorized based on root causes. Previous research on general Python library API misuse [230] notably omitted explicit specification of root causes in their classification framework. Existing research on the misuse of general Python libraries APIs [230] did not specify the root cause. However, they did identify a category that

Table 5.1: DL API misuse examples

Type		Examples	
Element	**Violation**	**Misuse**	**Fix**
Method	Missing	tensor_a	tensor_a.to(device)
	Redundant	embeds.detach().cpu().numpy()	embeds.detach().numpy()
	Replacement	tensor_a.int()	tensor_a.bool()
	Outdated	tf.scalar_summary('learning_rate', lr)	tf.summary.scalar('learning_rate', lr)
Parameter	Missing	torch.tensor(data_arg)	torch.tensor(data_arg, device=device_arg)
	Redundant	torch.zeros(arg_a, requires_grad=True)	torch.zeros(arg_a)
	Replacement	DenseLayer(arg_a,arg_b, act=tf.identity)	DenseLayer(arg_a,arg_b, act=None)
	Outdated	torch.chunk(inputs, arg_a, dim=1)	torch.chunk(self.inputs, arg_a, dim=1)
Condition	Missing	dtype = dtype_a	if isinstance(dtype, object): dtype = dtype_a
	Redundant	if self.flatten: x = x.flatten(1)	x = self.flatten(x)
	Replacement	if version < version_a:	if version < version_b:
	Outdated	if type == type_a:	if type == type_a or dtype == type_a

Table 5.2: Distribution of DL API misuse types

	Missing	**Redundant**	**Replacement**	**Outdated**
API Method	113 (46%)	138 (61%)	130 (62%)	74 (34%)
API Parameter	88 (36%)	56 (25%)	60 (29%)	115 (52%)
API Condition	43 (17%)	29 (17%)	17 (8%)	28 (13%)
Total	244	223	207	217

overlaps with the classification, which is the null reference check category versus the condition check. This category includes an if statement that checks whether a variable is null. Based on the unique feature of deep learning, this thesis identified 4 new categories, which are *Data conversion*, *Device Management*, *Algorithm*, and *Deprecation*. **API Misuse Symptom:** The severity of API misuses was classified based on their symptoms to understand their impact. Four symptom categories were identified: *Low Efficiency*, *Program Crash*, *Unexpected Output*, and *Return Warning*. It should be noted that *Low Efficiency* was also reported in an existing study [207].

5.3 API misuse in DL Libraries

5.3.1 Misuse Type

This thesis categorizes API misuse into two sub-dimensions, which are API violations (Section 5.3.1) and API elements (Section 5.3.1). Specifically, there are four types of API violations and three types of API elements. In total, 12 API misuse types are derived from considering different API violations and API elements. Table 5.2 shows the detailed distribution of API misuse in these two dimensions.

API Violation Type

API violations was categorized into four types, i.e., *Missing*, *Redundant*, *Replacement*, and *Outdated*. In general, the distribution of each violation is fairly even. The *Missing* violation indicates the absence of an API element. It contains 244 instances and is the most common violation type. Conversely, the *Redundant* violation entails a redundant API element. The redundant API method, with 138 instances, is the most common API misuse type among all 12 types. The *Replacement* violation involves replacing an existing API element, including replacing an existing API with a new one. API condition Replacement, with only 17 instances, is the rarest category among the 12 types. The *Outdated* violation represents a class change or renaming of the API elements. The difference between an *Outdated* and a *Replacement* lies in their semantics. An *Outdated* refers to a minor alteration that only affects the class or name while keeping the semantics almost the same, while a *Replacement* implies a significant modification to the API method.

API Element Type

This thesis categorized API elements into three types, i.e., *API Method*, *API Parameter*, and *API Condition*. An *API Method* refers to a deep learning API method involved in the API misuse with or without parameters. *API Parameter* refers to the parameters used in an API method. The value of the parameters can be defined in the API context or in the API method argument assignment. *API Condition* refers to the internal state of the program at runtime or condition before the API method call. To clarify each category's meaning, representative examples are presented in Table 5.1. Among the analyzed misuse patterns, API method emerges as the most prevalent element across Missing, Redundant, and Replacement categories. **Missing API Method** constitutes a distinct misuse type characterized by the absence of required API calls, representing a common implementation error. One common example in PyTorch is changing *tensor_a* to *tensor_a.to(device)* to move a tensor to a specific device. **Redundant API Method** denotes an API misuse caused by an unneeded API call. For example, in earlier versions of PyTorch (versions 0.4 and earlier), *cpu()* was used to convert a tensor into a CPU tensor. However, in a later version of PyTorch, *cpu()* is no longer required. **API Method Replacement** involves replacing one API call with another. For instance, in the PyTorch library, a mask tensor requires a boolean type. Thus, changing '.int()' to '.bool()' in 'tensor_a.int()' ensures that the tensor has a strictly boolean type.

Finding 1: The most common API element of API misuse is the *API Method* (455, 51.06%). The most common violation of API misuse is *Missing* (244,27.38%). The most common combination of violation and element is the *Redundant API Method* (138,61%) and the least common combination is *API Condition Replacement* (17,8%).

Table 5.3: Distribution of DL API misuse root causes

Root Cause Category	TensorFlow	PyTorch	Total
Data Conversion Error	72	174	246
Device Management Error	68	269	**337**
Algorithm Error	20	68	88
Deprecation Management Error	76	101	177
Null Reference Error	13	20	33
Other	2	8	10

5.3.2 Taxonomy of API Misuse Root Cause

This thesis conducted an investigation into the underlying root cause behind API misuses, categorizing them into five distinct types: *Algorithm Error*, *Deprecation Management Error*, *Data Conversion Error*, *Device Management Error*, and *Null Reference Error*. Table 5.3 shows the API misuse distribution of each root cause category. In general, *Device Management Error* with 337 instances is the most frequent root cause. On the other hand, *State Handling Error* and *Argument Error* are extremely rare in root causes.

Algorithm Error represents the math-related API misuses, typically involving errors such as division by zero or incorrect calculations. This type of error rarely exists in traditional libraries, but commonly exists in deep learning libraries due to the computation-intensive nature of deep learning libraries. For example, an API call missing the *eps* value as a parameter can result in a division-by-zero problem. By including the *eps* value (e.g., *eps = config.eps*) in the *nn.LayerNorm(dim)* API call, a small non-zero number is introduced to prevent division by zero. Identifying and addressing API misuse can be challenging, as it requires knowledgeable and experienced deep learning developers, which is often rare in the market.

Deprecation Management Error involves API misuses due to deprecated APIs or refactoring. API calls must be updated to accommodate the refactoring and version change, ensuring their continued usability. For instance, if one Script uses the *torch.nn.functional.softmax()* API and the other Script uses the *nn.functional-.softmax()* API, the API in another Script should be updated to match the one in the first Script. This standardizes API references and prevents potential program defects. Identifying API deprecation is one of the simplest forms of API misuse detection, as developers adhere to best practices by programming warning messages when deprecated APIs are used in the latest versions. Deprecation-related API misuse is also relatively easy to address, as modern integrated development environments (IDEs) support automated reference updates after refactoring. In fact, a considerable number of API misuse fixes are generated by code linting software. Although deprecation management may appear straightforward, several specific cases, such as condition checks, status updates, and parameter updates, can still be challenging.

Device Management Error pertains to the API misuse related to hardware and resource utilization. This category is particularly relevant to deep learning APIs due to

their distinct hardware utilization characteristics. Unlike traditional software libraries that assume CPU-only hardware and local execution, machine learning tasks are computationally intensive and primarily utilize GPUs or GPU clusters instead of CPUs. Therefore, machine learning libraries commonly offer GPU support and distributed computing capabilities. The assumption that the CPU is the sole hardware resource is no longer valid for machine learning libraries, leading to API misuses caused by incorrect hardware or resource configuration, as well as flawed assumptions about the hardware environment. An example of a common API misuse is the omission of the *device* parameter in an API call. In machine learning tasks, data objects must be stored in the appropriate hardware cache for accurate execution. Failing to provide the *device* parameter may result in incorrect hardware assignment of the data object, leading to program crashes.

Data Conversion Error relates to the incorrect shape or type of API input or output. Type problems are well-known issues in Python and other dynamic programming languages. In Python, variables lack specific types upon creation and may change types during runtime. While this design simplifies syntax to a great extent, it also introduces significant challenges. API calls often assume specific types for their variables, leading to failures when the parameter value provided to the API call does not match the assumptions. Developers already consider this problem when designing machine learning APIs and provide parameters that explicitly specify the type. For example, including the parameter $dtype = dtype$ in $tf.ones\_like()$ explicitly sets the return type of $tf.ones\_like()$ to *dtype*. On the other hand, Shape mismatch occurs often when passing variables to APIs. Executing deep learning models requires strict shape match when passing a tensor between layers. Each layer has its own constraint on the input tensor. Developers need to make necessary transformations to the tensor to make it compatible with the receiver. For instance, certain models require tensors to be flattened before passing them to the next layer. This problem can be resolved by adding the $.flatten(Tensor)$ API call. Shape mismatch is a unique type of API misuse in deep learning libraries because these libraries heavily rely on tensor computations.

Identifying data conversion API misuse can be challenging since the program might not immediately raise an error but rather produce incorrect results due to the incorrect calculation introduced by the API misuse. Detecting such API misuses requires a lot of experience in both deep learning libraries and algorithms. Given the absence of immediate errors, developers must manually review the code line by line to identify API misuses. While existing work has addressed tensor shape-related bug repair using static analysis approaches, there is still room for improvement in terms of efficiency, coverage, and ease of use.

Null Reference Error represents the null pointer exceptions-related API misuse. Null pointer exceptions are classic errors in computer programming. Thanks to the linting feature of modern IDEs, most API misuses are identified and rectified before merging the code into the project's codebase. Therefore, instances of such API misuse are rarely observed.

Others includes cases that may not belong to any of the predefined categories. One such example is argument error, which involves API misuses where necessary API arguments are missing. Instances of this type are infrequent since APIs typically require all necessary parameters, and omitting them would result in error messages.

Table 5.4: Distribution of DL API misuse symptoms

Symptom Category	TensorFlow	PyTorch	Total
Program Crash	97	226	**323**
Unexpected Output	57	153	210
Low Efficiency	63	218	281
Return Warning	30	34	64
Others	4	9	13

Finding 2: The most common root cause of API misuse in PyTorch is *Device Management Error* (269, 30.19%), while in TensorFlow, it is *Deprecation Management Error* (76, 8.52%). On the other hand, *Null Reference Error* shows the lowest number of API misuses in both libraries (13 in TensorFlow and 20 in PyTorch). Notably, *Device Management Error* (337), *Data Conversion Error* (246), and *Deprecation Management Error* (177) are the top three most common types of API misuse in both TensorFlow and PyTorch.

5.3.3 Taxonomy of API Misuse Symptom

In the sections below, DL API misuse instances are classified based on the symptoms they exhibited. The symptoms of API misuse were classified into four distinct categories, i.e., *Program Crash*, *Unexpected Output*, *Low Efficiency*, *Return Warning*, and others. Table 5.4 displays the instance count of each symptom. In the subsequent sections, each category will explained in detail.

Program Crash represents a category of API misuse that is relatively easy to identify. It is the most common symptom of API misuse. Instances falling under this category result in immediate program failures or crashes. For example, a previously functional program may fail to work properly after a version update or code refactoring due to inappropriate handling of the refactored API. The program crashes as a direct consequence of the failure to update the changes introduced in the API. Timely detection and rectification of such issues are vital to maintaining the stability and reliability of software systems that rely on the API. This category is the easiest to identify because it would crash the program immediately and throw an error with the line number.

Unexpected Output includes instances where developers deviated from the prescribed usage of the API, thereby employing an unexpected way of using API. Consequently, the output obtained differs from the expected result. This category has the most complex root causes because it does not closely relate to any specific root cause. Any root causes may result in an unexpected output as long as it does not throw an error. This type of symptom is hard to identify and debug since such a program does not throw an error but instead produces an incorrect result.

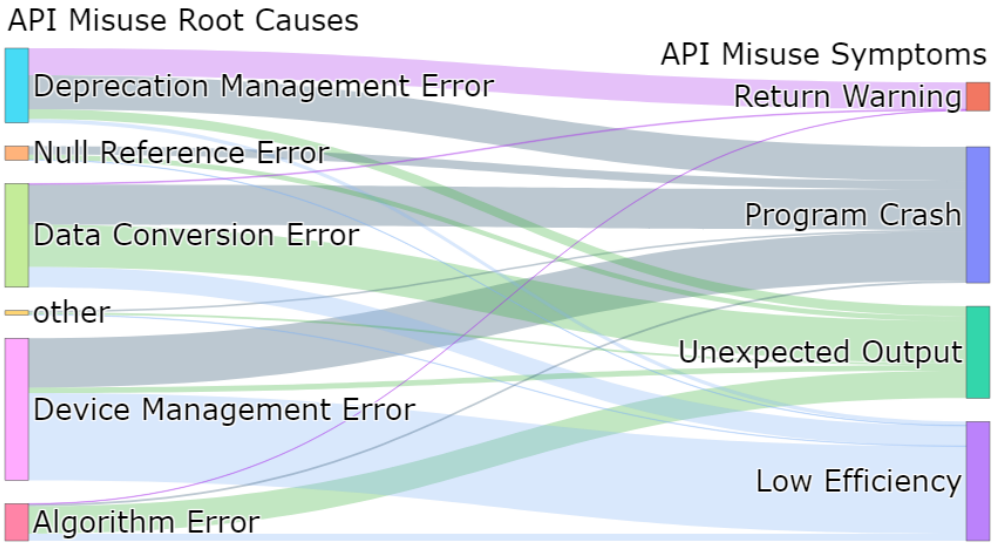

Figure 5.2: Mapping between root causes and symptoms

Low Efficiency refers to the slow execution of an API. It is closely related to device management errors because deep learning APIs can be accelerated by GPU devices. Failing to configure the device properly may very likely result in slow execution speed. Identifying errors related to low efficiency in API usage can be particularly challenging. Unlike other categories, low efficiency does not manifest as a warning or incorrect error. Instead, it produces a negative impact on the performance of the program, resulting in decreased efficiency or slower execution. Inexperienced developers may not notice the error because they may not have enough experience to have a proper expectation of the performance[233]. Such issues often require careful analysis of performance bottlenecks to pinpoint and resolve the underlying problems. In the deep learning industry, identifying these inefficiencies and optimizing the code is crucial to ensure the overall performance and responsiveness of the application or system utilizing the API.

Return Warning refers to instances where the API usage returns warnings instead of errors. Return warnings typically occur when developers use deprecated APIs in a newer version of the software. The impact of such warnings on the program's functionality depends on the actions taken by the library developers. Some developers provide backward compatibility APIs, allowing the continued use of deprecated functionalities. However, in some cases, library developers discontinue support for deprecated APIs in newer versions. As a result, developers need to heed these warnings seriously and take appropriate actions, such as updating their code to utilize alternative APIs or adopting compatible replacements, to ensure the smooth operation and maintainability of their software.

Others The **Others** designation applies to cases where insufficient evidence exists to determine specific symptoms conclusively. These indeterminate instances are systematically categorized under the 'Others' classification.

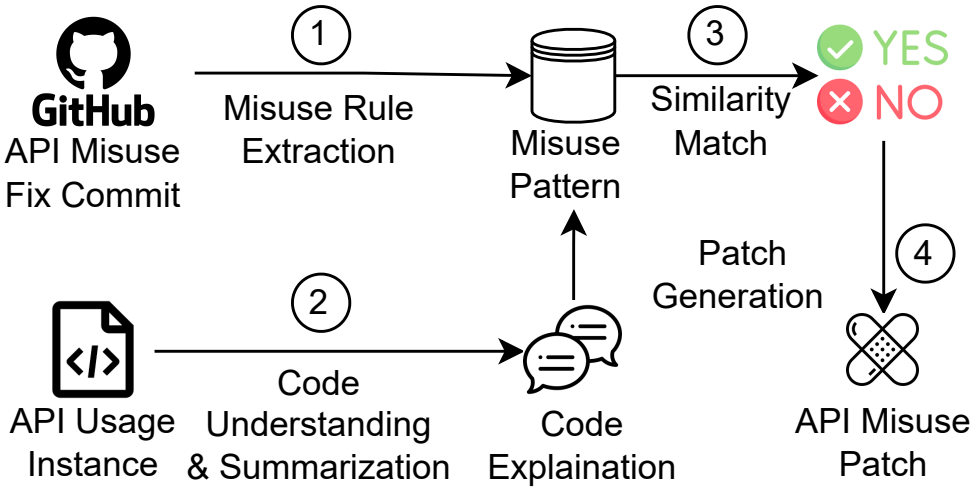

Figure 5.3: Overview of our LLM-based DL API misuse detector

A Sankey graph was created to illustrating the relationship between root causes and symptoms to interpret the outcomes of API misuses, as shown in Figure 5.2. The graph clearly demonstrates a correlation between the root cause and the resulting symptom. For example, the majority of low-efficiency symptoms stem from device management errors, while nearly all return warning symptoms are caused by depreciation management errors.

Finding 3: The most common symptom of DL API misuse in PyTorch and TensorFlow is *Program Crash* (226 for PyTorch and 97 for TensorFlow). On the other hand, *Return Warning* shows the lowest number of API misuses in both libraries (30 in TensorFlow and 34 in PyTorch).

5.4 LLMAPIDet

As shown in Section 5.3, DL API misuse primarily involves issues related to data and device usage, such as returning incorrect float types or accidentally running GPU tasks on the CPU. These issues are beyond the detection scope of most of the existing API misuse detection tools, which rely on the matching of errors or warnings to trigger the detection process. Detecting such misuse often relies on manually created API rules by experienced experts, which can be both costly and impractical in terms of scalability.

Inspired by the recent great progress of LLMs (Large Language Models) in code understanding and various source code-related tasks [217, 234, 235], this thesis proposes LLMAPIDet , the first LLM-based DL API misuse detector for automated misuse rule extraction, API misuse detection, and patch generation. Note that the study is based on ChatGPT [216] as it

Table 5.5: Prompt template for extracting misuse rules.

"prompt": Please identify the rules for fixing the API Method problem in the following code change. Example One: {code_removed_one} {code_added_one} {misuse_rule_one} Example Two: {code_removed_two} {code_added_two} {misuse_rule_two} Question: {code_removed}{code_added} **"output"**: {misuse_rule}

has shown great potential in software engineering, e.g., code generation [217], bug repair [218], and program understanding [219], among comparable LLMs.

5.4.1 Approach

The overview of the proposed tool is illustrated in Figure 5.3, comprising four steps.
in step one, A knowledge base of misuse rules was constructed based on the 891 confirmed API misuses identified during the empirical analysis. The process involved extracting API misuse rules using ChatGPT to analyze source code and summarize the corresponding patterns. Table 5.5 presents the prompt template for misuse rule extraction. The template includes: (1) a task description, and (2) two manually created demonstration examples containing: (i) pre-fix code (removed content), (ii) post-fix code (added content), and (iii) corresponding manually defined misuse rules. The extraction process followed a consistent methodology: for each code change, the 'code removed' and 'code added' components were systematically identified. This approach was then applied uniformly across all 891 confirmed API misuses, with ChatGPT generating corresponding outputs using the same prompt structure. For conciseness, Table 5.5 displays only the template while omitting specific example text.
In step two, given a new API usage instance, ChatGPT was utilized to generate a code explanation. Table 5.6 shows the template for generating code explanations. The prompt consists of an instruction that prompts ChatGPT to describe a given piece of code snippet, along with a "code_snippet" variable. For each ChatGPT API call, the "code_snippet" variable is filled with the API usage instance along with its context. The context length is set similarly to existing GitHub historical commit studies [236, 237, 238], which is 4 lines both above and below the API usage instance.
In step three, ChatGPT determines the applicability of any misuse rules to the given API usage instance. If an API usage instance satisfies the conditions defined by the misuse rules, it is considered a detected API misuse. The prompt for ChatGPT in this step is constructed

Table 5.6: Prompt template for generating code explanation

"prompt": Please describe what the following code snippet does in two sentences: Code snippet: {code_snippet} **"output"**: {Answer}

Table 5.7: Prompt template for API misuse detection

"prompt": Please read the following code snippet and API misuse rules. Then, answer whether the misuse rules can be applied to the code snippet. If the pattern can be applied, answer "Yes"; if not, answer "No". Code snippet: {code_snippet} Misuse rules: {misuse_rules} **"output"**: {Decision}

by concatenating the API usage instance and the list of misuse rules filtered by the API usage method name. To identify potential misuse rules that best match the given instance, the misuse rule list is re-ranked by cosine similarity between the code explanation obtained in step two and each misuse rule in the knowledge base. This approach is adopted because suboptimal cosine similarity was observed between source code embeddings and misuse rule embeddings, resulting from the representation gap between source code and plain English text. To address this issue and achieve better alignment, the source code was replaced with its corresponding English description, resulting in an enhanced cosine similarity calculation. After the re-ranking, the top 4 misuse rules are selected as candidates. The number of rules is set to 4 to ensure that the total prompt length remains within the token limit of ChatGPT. This composed prompt is then fed into ChatGPT to query whether any of the identified misuse rules can be applied to the provided example. This step results in a response of either 'YES' or 'NO', indicating the presence or absence of API misuse in the code. Table 5.7 shows the template for detecting API misuses in this step.

In step four, for each API misuse instance identified in step three, ChatGPT performs a structured reasoning process. Following the prompt template, it first analyzes the problem through logical steps before generating a potential patch according to the relevant misuse rule. The thinking steps and patch are generated following a predefined output format. The prompt template for generating the patch is shown in Table 5.8.

Table 5.8: Prompt template for patch generation

"prompt":
Please read the following code snippet and misuse rules. Then, think step by step and generate a patch for the code snippet. Please ignore any indentation problems in the code snippet. Fixing indentation is not the goal of this task. If the pattern can be applied, generate the patch.
Code snippet: {code_snippet}
Misuse rules: {misuse_rules}

"output": {Think steps}{Patch}

Table 5.9: Performance of LLMAPIDet in detecting API misuse

	Exp. One		Exp. Two		
	Detected	Recall	Detected	Verified	Precision
LLMAPIDet	48	16.49%	368	119	32.33%
TADAF	0	-	41	5	12.2%

Table 5.10: Performance of LLMAPIDet in patch generation

		Generated	Verified	Accuracy
Exp.One	LLMAPIDet	48	10	22.83%
	TADAF	0	-	-
Exp.Two	LLMAPIDet	119	46	38.65%
	TADAF	5	5	100%

5.4.2 Experiment Setting

Two experiments were designed to evaluate the performance of LLMAPIDet and the benchmark approach TADAF. Due to the quote limitation of ChatGPT 4.0 at the time of conducting the experiments, GPT 3.5 was used for the experiments. **In the first experiment**, LLMAPIDet was built using 600 randomly selected instances from the 891 confirmed API misuses, with the remaining 291 API misuses reserved as the testing dataset (i.e., Experiment One). Both tools were tested on Experiment One to detect API misuses and compare their recall rates.

Additionally, a second experiment was designed to address concerns about memorization in LLM-based approaches [221, 220], where correct predictions might be based on publicly available online answers. To mitigate the memorization issue, **in the second experiment**, 4,359 API usage instances were collected from the latest version of 10 open-source GitHub

projects (i.e., Experiment Two) other than the 200 projects used in the empirical study (see Section 5.3). All of these projects are actively maintained, as their latest updates range from July 25, 2023, to July 29, 2023, which is after the timestamp of the live version of ChatGPT. The number of API usage instances in these projects ranges from 78 to 2,131, with an average count of 436 instances. The dataset covers a total of 204 unique APIs. The following text demonstrates how much API misuse can be detected in Experiment Two, with manual verification of its validity using both tools.

5.4.3 Performance of LLMAPIDet

The results on the detection of DL API misuses are presented in Table 5.9. In experiment one, LLMAPIDet successfully detected 48 API misuses, whereas TADAF could not detect any instances. The recall values are 16.49% and 0% for LLMAPIDet and TADAF, respectively. In the experiment two dataset, LLMAPIDet identified 368 instances of API misuse, while TADAF only found 41 misuse instances. Each instance was manually examined, confirming 119 API misuses reported by LLMAPIDet and 5 misuses detected by TADAF as true bugs. The precision of LLMAPIDet is 32.33%, while the precision of TADAF is 12.2%.

The results of patch generation are shown in Table 5.10. In the first experiment, LLMAPIDet generated patches for 48 API misuses, while TADAF did not generate any patches. The accuracy of LLMAPIDet is 22.83%. In the second experiment, LLMAPIDet generated patches for 119 API misuses, while TADAF generated patches for 5 instances. Each patch was manually investigated, confirming 46 correct patches for LLMAPIDet and 5 patches for TADAF. The accuracy of LLMAPIDet is 38.65%, while the precision of TADAF is 100%. To validate the findings, the API misuse instances were reported to the developers of these projects by creating GitHub issues.

The following example demonstrates a successfully detected API misuse. In line 3, the input tensor should be assigned to a device to avoid potential device management errors when the args.mode is not equal to 'gpu' in line 5. LLMAPIDet detects this API misuse and suggests a fix (i.e., assigning the tensor to a device) in line 9, which has been verified as correct.

```
1 # LLMAPIDet detects API Misuse in the code below:
2 if args.bbox_init == 'two':
3     input = transform(img_step2).unsqueeze(0)
4     with torch.no_grad():
5         if args.mode == 'gpu':
6             input = input.cuda()
7 # Patch:
8 if args.bbox_init == 'two':
9     input = transform(img_step2).unsqueeze(0).to(device)
10    with torch.no_grad():
11        if args.mode == 'gpu':
12            input = input.cuda()
```

The following example illustrates a case in which LLMAPIDet fails to detect API misuse. The problem arises from the presence of incomplete code in the last line of the context, leading ChatGPT to generate code to complete the line instead of primarily focusing on examining and editing the existing code for misuse detection.

```
# LLMAPIDet fails to detect API Misuse in the code below:
with tf.name_scope('distribution_errors'):
    ...other code...
    tf.compat.v2.summary.scalar(
# Patch:
with tf.name_scope('distribution_errors'):
    ...other code...
    tf.compat.v2.summary.scalar('mean',
    tf.reduce_mean(distribution_errors),
    step=self.train_step_counter)
```

One way to address the problem above is to gather more context for the code snippet. However, this can be challenging as it is difficult to determine the appropriate amount of context to provide. If too much context is given, irrelevant information may be included, resulting in noise. Although existing methods like code slicing can help alleviate the issue, achieving high-quality code slicing in Python is not a straightforward task.

5.5 Evaluation

5.6 Performance of SOTA on DL API misuse detection

To assess the effectiveness of the existing state-of-the-art API misuse detector, i.e., *TADAF* [207], in detecting DL API misuses, the tool was run on the dataset that contains 891 DL API misuses. Please note that the replication package of *TADAF* is not publicly available. Despite multiple attempts to contact the author for the latest version of the replication package, no response was received. Consequently, the replication of *TADAF* was conducted to the best possible effort. The process of *TADAF*, as described in the paper, was rigorously followed to replicate the tool based on their description. Specifically, *TADAF* utilizes a keyword-matching mechanism based on the 11 API misuse patterns to detect API misuse.

To ensure the accuracy of the replication, the experiments were conducted on the same dataset used in the original evaluation of *TADAF*. This thesis's replication reports the same set of bugs as *TADAF* in its experimental projects, which confirms the correctness of this thesis's replication.

For the evaluation of *TADAF*, the replication was applied to the DL API misuse dataset containing 891 confirmed API misuse instances. The results indicate that *TADAF* only detected 3 out of 891 API misuse instances. *TADAF*'s low performance can be attributed to its limited API misuse patterns used to find API misuses, while mining patterns demand

non-trivial manual effort from expert developers to identify, summarize, and validate each misuse pattern.

5.7 Discussion

5.7.1 Difference between DL API misuses and API misuses of traditional software

Existing studies on API misuse primarily focus on general software with programming-related categories, such as synchronization, control flow, and state handling [206, 205, 59]. DL API misuses exhibit distinct characteristics, such as tensor-related API misuse and resource-related API misuse. These differences arise from the unique design, problem-solving paradigm, knowledge representation, and computationally intensive nature of DL APIs. The following sections delve into a detailed description of these contrasting characteristics.

Differences in data type-related API misuse: This thesis's study confirms that DL libraries also experience type-related issues. However, a few differences were identified compared to traditional libraries. For instance, when comparing the return value of a torch tensor to a number, developers may need to cast the number as a tensor using *torch.tensor()* to match the API's return type; otherwise, the comparison may fail. However, value comparison in traditional Python programs often allows comparison between different data types without specifically casting one type to another. Additionally, types are sometimes intentionally adjusted to accommodate business logic. For example, developers may cast float64 numbers to float32 to optimize computation resources and expedite calculations at the cost of slight precision loss.

Shape and algorithm-related API misuse: API misuses can arise from incorrect shape assumptions or erroneous algorithms. These types of API misuse are relatively uncommon in traditional API misuse as they typically do not involve tensor computations. Identifying and locating these types of errors can be challenging since they often do not cause immediate program crashes. For example, if an API expects a 3x2 shape input tensor (e.g., [1,2],[3,4],[5,6]), but a 2x3 shape tensor is provided (e.g., [1,2,3],[4,5,6]), the appropriate API misuse fix would be to invoke *Tensor.transpose()* to transpose the shape. However, developers without enough mathematical knowledge might instead use *Tensor.reshape()* to reshape the tensor to an incorrect shape (e.g., [1,4],[2,5],[3,6]), resulting in incorrect data. While this API misuse may not trigger an immediate error, it eventually leads to incorrect results and difficulties in tracing the root cause. Given their lack of transparency and intricacy, these bugs are considered the most challenging to handle. Prior research has also highlighted similar issues in machine learning defects[239].

Resource and hardware-related API misuse: A significant number of API misuses related to resource management (37.8%) can be observed. Given the computational intensity of machine learning tasks, GPU acceleration and distributed computing are commonly employed. As a result, API execution shows inconsistencies depending on resource availability and environment configuration. For example, tensors involved in the same calculation process

must reside on the same hardware device; otherwise, the program crashes when it fails to locate the tensor on the assumed device. Other resource-related misuses include setting the GPU as the default option on CPU-only devices and encountering missing GPU support libraries when switching between different GPU models.

Versioning-related API misuses: Machine learning libraries experience frequent refactoring and version updates compared to other libraries. While traditional software design builds on top of established computer science theories and software development best practices, the rapidly evolving nature of machine learning theory introduces new concepts regularly. The rapidly evolving nature of machine learning libraries introduces many refactoring-related API misuses. This thesis's observations revealed three levels of refactoring-related API misuse fixes in PyTorch and TensorFlow, ranging from easy to challenging. The first level involves class-level refactoring, where the class of the API changes while leaving the API method unchanged. This is a relatively straightforward update that can be automated. The second level involves refactoring both class and method names, requiring manual effort to create a mapping between the old and new APIs. Despite the additional complexity, this type of update remains feasible due to the guaranteed one-to-one mapping. The third and most challenging level involves library redesign, where numerous APIs are removed, functionalities are split into multiple new APIs, or certain functionalities are discontinued. In such cases, developers may need to learn the new library and rewrite their code as an alternative API as refactoring is not guaranteed.

5.7.2 Guidelines for avoiding DL API misuse

Variable type enforcement: The analysis suggests incorporating type annotations and type assertions into APIs and API parameters to mitigate type errors. By explicitly specifying the expected types, developers can catch type-related issues early in the development process. Type annotations also facilitate more transparent type management in Python, as they serve as documentation for both humans and automated tools. Additionally, integrating code lint checkers like PEP8[240] into the development pipeline is recommended to enforce consistent coding conventions and improve code quality.

Resource availability checking: To ensure efficient resource and environmental management in machine learning applications, implementing a global configuration file that is consulted before executing any business logic is recommended. Currently, machine learning library APIs often require explicit device specifications, which are handled individually by each API. By centralizing device management in a dedicated layer before the business logic layer, one can ensure consistent resource and environment assumptions for all resource-sensitive machine learning APIs. This approach simplifies the process of managing resources such as GPUs and facilitates seamless execution across different hardware configurations. Applying techniques like environment detection and configuration generation further enhances the reliability and reproducibility of resource-sensitive machine learning APIs and workflows.

API deprecation handling: Although developers already make efforts to handle API deprecation by issuing warnings and maintaining backward compatibility, unforeseen risks can

still arise. As deep learning theory and practices evolve, popular architectures may become suboptimal, requiring a redesign of the library structure to meet new technical requirements. Such redesigns and refactorings are often unavoidable. In some cases, the redesigned library may not provide alternative APIs, leaving developers to face the challenge of rewriting their code using the new version. This task can be frustrating, and developers may struggle to find comprehensive guidelines and solutions for version migration. To enhance the robustness of deprecation support, it is crucial to provide not only thorough documentation but also automated API mapping mechanisms. These mechanisms can aid in transitioning existing codebases to new versions by suggesting equivalent replacement APIs and minimizing the effort required for code migration. By improving archive code support, developers can more effectively adapt to evolving libraries while maintaining code compatibility and minimizing disruptions to their projects.

5.8 Related Work

An API misuse refers to the incorrect usage of an Application Programming Interface (API), where there are violations of the API's usage constraints, such as call order or preconditions [206]. These misuses can lead to various issues, including software crashes, bugs, data loss, and vulnerabilities.

Most of the existing API misuse studies mainly focus on Java projects. Amann et al. [205] presented the API misuse dataset, i.e., MUBENCH, which contains manually identified 89 API misuses from 33 general Java projects. Later, they further presented a systematic evaluation of existing API misuse detectors for Java [59].

Recently, Baker et al. [207] conducted an API misuse study on TensorFlow API misuses and also proposed an API misuse detector for TensorFlow APIs. They performed a manual investigation on a limited scale of 15 Tensorflow bugs and identified 7 API misuse patterns based on existing datasets from two empirical studies on DL program bugs [210, 211]. Manual investigation of over 4,000 API misuse instances confirmed 891 API misuses in PyTorch and TensorFlow APIs. To the best of this thesis's knowledge, this represents the first large-scale API misuse study exclusively focused on DL libraries. Li et al. [230] performed a large-scale study on API misuse for over 500,000 bug fixes and classified them into 9 categories for general Java Projects. Their work focuses on general API misuse in open-source Java projects, whereas this study examines Python Deep Learning APIs. The findings reveal several significant differences in misuse categories and patterns between these domains. Wan et al. [209] conducted an API misuse study on machine learning cloud service APIs for 4 major service providers on 3 primary application types. Compared to this thesis, they focus on the cloud service APIs from commercial service providers such as Microsoft Azure and 3 application types, such as vision, language, and speech, while this thesis focuses on open-source backbone deep learning libraries APIs in PyTorch and TensorFlow.

5.8.1 LLM in Software Engineering

Recently, the field of software engineering research has witnessed a significant impact with the emergence of Large Language Models (LLMs) [84]. Chen et al. [217] introduced Codex, the first GPT-based LLM for source code, which subsequently served as the foundational model for code-related research. Codex has found applications in multiple software engineering domains, including automated program repair (Prenner et al. [234] and Sobania et al. [241]), automated test case generation (Xie et al. [235] and Yuan et al. [242]), and vulnerability detection (Cheshkov et al. [243] and Nair et al. [244]).

Despite the impressive code generation abilities exhibited by LLMs, researchers have begun to evaluate the quality of the generated code from multiple perspectives, including correctness (Liu et al. [245]), performance (Feng et al. [246]), and security (Khoury et al. [247]). Prompt engineering is the primary approach for building LLM-based applications. Previous work has studied the methodology of prompt tuning (Zhang et al. [248] and Shrivastava et al. [249]) specifically in the context of software engineering. In this thesis, this thesis designs and refine prompts through iterative testing and evaluation on a small dataset.

5.9 Threats to Validity

Construct Validity: In the empirical study, this thesis focused exclusively on Python deep learning libraries, excluding deep learning libraries in R or Java due to differences in programming language characteristics. To determine the number and names of categories, this thesis conducted a preliminary analysis based on 200 randomly selected examples to determine the appropriate granularity and category names. *TADAF* was selected as the benchmark tool due to its high relevance to this thesis. Unlike other tools, such as *Ariadne* [250], which serve as general bug detectors for DL libraries, *TADAF* specifically addresses DL API misuse detection. This specialization makes *TADAF* the most suitable choice for this thesis. Although *TADAF* is designed for TensorFlow, some of its rules are still suitable for PyTorch.

Internal Validity: The manual labeling process required significant effort. This thesis ensured accuracy by having multiple rounds of discussion to establish a consensus on the labeling standards for 200 randomly selected examples. The rest of the labeling work was then assigned to several experienced Ph.D. students. However, it is important to recognize that this approach may impact the overall accuracy of the classification.

External Validity: During the evaluation of the detector, the non-deterministic nature of LLMs introduced variability in the output of the LLM model. Consequently, these differences could propagate through subsequent interactions with the LLM in the detector workflow, leading to different results. Employing a fully open-sourced model such as LLaMA [251] with a frozen seed may enhance reproducibility.

5.10 Conclusion

This thesis presented the first extensive study on API misuse within two major DL libraries, i.e., TensorFlow and PyTorch. This thesis identified 891 API misuses from the 200 most starred projects on GitHub built with the two libraries and provided insights into their categories, root causes, and symptoms. To address the challenges in detecting DL API misuses, this thesis introduced LLMAPIDet , a novel LLM-based tool for DL API misuse detection and patch generation. This thesis's evaluation demonstrated the effectiveness of LLMAPIDet . The contribution of this work is, to the best of this thesis's knowledge, this is the first large-scale analysis to demystify and detect DL API misuses in PyTorch and TensorFlow. A benchmark containing 891 instances of DL API misuse was created, with the dataset and source code publicly released for replication. Future work will focus on expanding LLMAPIDet 's capabilities and exploring additional automation strategies for DL API misuse detection.

Chapter 6

Conclusion

6.1 Thesis Findings and Contributions

This thesis has tackled the challenge of enhancing API usability and reliability, with a focus spanning traditional software development and deep learning frameworks—both critical to the creation and deployment of complex systems. By systematically exploring these diverse contexts, this research bridges foundational software engineering practices with the evolving needs of deep learning applications. The findings and contributions significantly advance the state of API utility, offering innovative solutions that empower developers and elevate standards in both traditional and emerging software development paradigms.

6.1.1 Key Findings

1. **Complexity of ML APIs**: The research confirmed the inherent complexity of ML APIs and their impact on developer efficiency and system reliability. The findings underscored the necessity for advanced tools that can interpret the context of API usage to provide more accurate recommendations and detect misuse.

2. **Inadequacy of Existing Tools**: The analysis revealed that current API recommendation systems and misuse detection tools are limited in their ability to handle the dynamic and contextual nature of ML workflows. This inadequacy often results in developers resorting to trial-and-error methods, leading to increased development time and potential system failures.

3. **Effectiveness of Advanced Techniques**: The empirical studies conducted within this thesis demonstrated that incorporating machine learning techniques, particularly contrastive learning and natural language processing (NLP), into API recommendation systems significantly improves the accuracy of recommendations and the detection of misuse.

4. **Practical Integration and Impact**: The integration of the proposed framework into development environments, including Integrated Development Environments (IDEs) and Continuous Integration/Continuous Deployment (CI/CD) pipelines, has shown promising results in reducing API-related errors and improving developer productivity.

6.1.2 Major Contributions

1. **Framework for ML API Usability**: This thesis introduced a comprehensive framework aimed at analyzing and enhancing the usability of ML APIs. The framework provides a structured approach to understanding API design, developer interactions, and common usage patterns, which can guide the creation of more intuitive and error-resistant APIs.

2. **Advanced API Recommendation System**: A novel API recommendation system leveraging contrastive learning was developed. This system offers precise, context-aware API suggestions by utilizing contextual code embeddings and semantic understanding, significantly reducing the likelihood of incorrect API usage.

3. **Automated API Misuse Detection and Correction**: An automated tool for detecting and correcting ML API misuses was designed and implemented. This tool uses static code analysis and runtime monitoring to identify common API misuse patterns and can perform automatic corrections, enhancing development efficiency and system stability.

4. **Taxonomy of ML API Misuse Patterns**: A taxonomy of common ML API misuse patterns was constructed, providing insights into the root causes of API misuse and assisting developers and API designers in mitigating errors and improving API design.

5. **Evaluation and Validation**: The effectiveness of the proposed recommendation and misuse detection tools was evaluated through comprehensive experiments and user studies, demonstrating a significant reduction in API misuse rates, faster debugging times, and improved system performance.

6. **Open-Source Contribution**: By open-sourcing the API recommendation and misuse detection tools, this research contributes to the broader ML and software engineering communities, enabling widespread adoption and continuous improvement of these tools.

In conclusion, this thesis has made substantial contributions to the field by not only identifying the challenges associated with ML API usability and reliability but also by proposing and validating novel solutions that leverage advanced machine learning techniques. These contributions have the potential to significantly impact the way ML APIs are developed and used, leading to more efficient, reliable, and scalable ML software development.

6.2 Future Work

While this thesis has made significant strides in addressing the challenges of machine learning (ML) API usability and reliability, there are several avenues for future research and development that can build upon the findings and contributions presented herein.

6.2.1 Enhancing the Recommendation System

Expanding the Scope of APIs: The current recommendation system is limited to a specific set of ML APIs and frameworks. Future work could aim to broaden the scope to include APIs from traditional software development environments, cloud platforms, and emerging technologies like edge computing, IoT, and augmented reality. This expansion

would significantly enhance the system's utility by supporting a diverse range of development scenarios and use cases. Furthermore, incorporating APIs from low-code and no-code platforms would make the recommendation system more accessible to a wider audience, including non-technical users. Integrating APIs from various programming languages and platforms would also support cross-platform development, where applications utilize multiple services and systems that interact dynamically. This would ultimately result in a more versatile and universally applicable recommendation system that serves developers working in various domains, such as finance, healthcare, automotive, and entertainment.

Incorporating More Contextual Factors: While current recommendation systems focus primarily on the API's functionality, there is a growing need to integrate more granular contextual factors into the recommendation algorithm. Future work could include incorporating project-specific constraints, such as resource limitations (e.g., memory, computational power), compliance requirements (e.g., data privacy regulations), and security concerns (e.g., secure data handling). Furthermore, understanding developer preferences—such as coding style, preferred libraries, or frequently used APIs—could result in more personalized and accurate suggestions. Environmental variables like the specific operating system, runtime environment (e.g., local development vs. cloud), and even hardware configurations (e.g., GPUs, edge devices) should be factored in to ensure that the API recommendations align with the developer's operational context. In addition, integrating dynamic contextual information, such as changes in the development lifecycle, dependencies between modules, and integration with external systems, could refine recommendations in real-time, adapting to evolving project needs. This holistic view of context would improve the precision of API recommendations, reduce the risk of misuse, and ultimately accelerate the development process.

Real-Time Recommendation Engine: A real-time recommendation engine would provide instant, in-context feedback as developers write code, allowing them to make decisions on-the-fly. This would significantly improve developer productivity by reducing the need for external documentation lookups or trial-and-error testing. Building this real-time system would require a deep integration with Integrated Development Environments (IDEs) and version control systems (e.g., Git) to track code changes and provide immediate suggestions based on the evolving project context. Leveraging techniques such as natural language processing (NLP) for code understanding and reinforcement learning for adaptive recommendation generation could enable the engine to deliver increasingly relevant suggestions over time. Additionally, incorporating an automatic error detection system that analyzes code as it is written, highlighting potential issues before execution, would be a valuable addition. This could include not only syntax errors but also logical errors, inefficient API usage, or compatibility issues with other parts of the code. Real-time feedback would empower developers to resolve issues early in the development cycle, improving software quality and reducing debugging time. As AI-powered code assistants become more sophisticated, integrating real-time APIs with these tools would provide seamless collaboration between developers and AI, fostering a more intuitive and efficient development workflow.

6.2.2 Improving Misuse Detection

Advanced Anomaly Detection Techniques: The growing complexity of modern software and ML workflows demands more sophisticated approaches to anomaly detection in API usage. Future research could explore employing advanced machine learning models, such as graph neural networks (GNNs) or temporal convolutional networks (TCNs), to capture intricate relationships and dependencies between API calls, especially in multi-API workflows. Additionally, integrating unsupervised learning methods, such as variational autoencoders (VAEs) or contrastive learning techniques, could improve the detection of subtle or previously unseen misuse patterns without requiring extensive labeled datasets. Another promising direction is the incorporation of ensemble anomaly detection systems, where multiple models are combined to enhance robustness and reduce false positives. For real-time applications, streaming anomaly detection algorithms could be adapted to monitor API behavior dynamically, enabling the identification of misuse in fast-evolving systems such as edge computing, real-time analytics, or autonomous systems. By adopting these advanced techniques, future tools can better accommodate the complexities and variability of modern development environments.

Automated Root Cause Analysis: While current systems focus primarily on detecting API misuse, future tools could provide developers with automated root cause analysis to explain why a particular API call is flagged as misuse and how to resolve the issue. Such tools could leverage causal inference techniques to trace errors back to their origin, identifying the specific parameters, configurations, or sequences that triggered the anomaly. For instance, tools could analyze call stacks, dependency graphs, or execution traces to pinpoint problematic interactions between APIs or with external resources. Integrating explainable AI (XAI) methods would also enhance transparency, providing developers with clear, human-readable explanations of the underlying issues. These insights could be further enriched with contextual recommendations for fixing the problem, such as suggesting alternative configurations, compatible parameters, or best practices derived from similar use cases. Additionally, root cause analysis tools could incorporate feedback loops, allowing developers to confirm or refine the suggested cause, thus improving the system's accuracy over time. By equipping developers with actionable insights into API misuse, these tools could significantly reduce debugging time, improve code quality, and enhance trust in the API recommendation and error detection system.

6.2.3 Broader Impact and Societal Considerations

Ethical API Usage: Future research could address the ethical implications of API usage, ensuring that the recommendations and detections align with ethical standards and best practices. This includes incorporating mechanisms to prevent bias in API suggestions, particularly in sensitive domains like hiring, healthcare, and finance, where unethical or biased API usage could have significant societal consequences. Additionally, transparency in how API recommendations are generated should be prioritized, using techniques such as explainable AI (XAI) to provide developers with clear, interpretable insights into the

system's decision-making processes. The research could also explore guidelines and tools to promote responsible API usage, such as flagging APIs associated with potential misuse, harmful outcomes, or violations of privacy regulations like GDPR or HIPAA.

Inclusivity and Accessibility: Ensuring that the tools developed are accessible and usable by a diverse range of developers, including those with disabilities, could be an important area of focus. This could involve adhering to web content accessibility guidelines (WCAG) and developing user interfaces that accommodate assistive technologies, such as screen readers, voice input systems, and alternative input devices. Furthermore, language accessibility should be addressed by providing support for multiple languages in documentation, tutorials, and API recommendations. Research could also explore ways to make the tools more intuitive for developers with varying levels of expertise, including beginners and those from non-technical backgrounds. By focusing on inclusivity and accessibility, these tools can empower a broader audience to leverage APIs effectively, promoting diversity and equity in software development.

6.2.4 Cross-Domain Applications

Application in Other Domains: The techniques and tools developed in this thesis could extend beyond the realm of machine learning to other domains such as web development, mobile app development, game development, and IoT. For instance, in web development, these tools could assist developers in selecting optimal APIs for front-end frameworks, server-side programming, or database management. In mobile app development, they could enhance the usability of APIs for platform-specific features like geolocation, notifications, and hardware integration. By adapting the methods to these contexts, the usability and reliability of APIs across diverse software ecosystems could be significantly improved, fostering innovation and reducing development overhead in multiple industries.

Cross-Disciplinary Collaboration: Encouraging collaboration between computer scientists, cognitive psychologists, and user experience (UX) designers could lead to the creation of more intuitive and user-friendly API tools. Cognitive psychologists could provide insights into how developers process and learn API documentation, while UX designers could focus on creating interfaces and recommendation systems that align with human cognitive processes and workflows. Additionally, sociological and educational perspectives could inform strategies for making APIs more approachable for novice developers, such as through gamified learning experiences or adaptive tutorial systems. These interdisciplinary efforts could result in tools that are not only functional but also align with the natural behavior and preferences of developers.

In conclusion, while this thesis has laid a strong foundation for enhancing ML API usability and reliability, the future work outlined above holds the potential to extend these advancements into other domains and foster innovative cross-disciplinary approaches. These efforts could revolutionize the way developers interact with APIs, leading to more efficient, effective, and error-free software development practices across a wide range of applications.

6.3 Closing Remarks

As this research journey concludes, it is essential to reflect on the significance of the work conducted and its implications for the field of machine learning (ML) and software engineering. This thesis has endeavored to bridge the gap between the complexities of ML APIs and the practical needs of developers, with the ultimate goal of enhancing the efficiency, reliability, and scalability of ML software development.

Over the course of this study, this thesis has explored the challenges inherent in ML API usability and reliability, delving into the depths of developer intent, API documentation ambiguity, and the limitations of current recommendation and misuse detection tools. The findings have shed light on the critical need for advanced systems that can provide context-aware, semantically rich recommendations and detect misuse with high precision.

The contributions made in this thesis, including the development of an ML API usability framework, a novel API recommendation system using contrastive learning, and an automated API misuse detection and correction tool, represent a significant step forward in addressing these challenges. These tools and frameworks have been designed with the developer in mind, aiming to reduce the cognitive load associated with API usage and to minimize the risk of errors that can have far-reaching consequences.

The empirical studies and user evaluations conducted have demonstrated the effectiveness of the proposed solutions, showing a marked improvement in recommendation accuracy, error detection rates, and overall developer productivity. These results are not just numbers on a page; they represent real-world benefits to developers and organizations alike, from reduced debugging times to improved system robustness and, ultimately, to more reliable ML models that can safely and effectively be deployed in critical applications.

However, as with any research, this work is not the end but just a start. The future work outlined in the previous section presents a roadmap for continued innovation and improvement. The integration of emerging technologies, the expansion of the recommendation system's scope, and the exploration of ethical and societal considerations are just a few of the many exciting avenues for future research.

In closing, I would like to express my gratitude to all those who have supported this research, including my advisors, colleagues, and the broader academic community. Your insights, feedback, and collaboration have been invaluable in shaping this work. To the developers who will use the tools and frameworks developed in this thesis, I hope they serve you well in your quest to create powerful, reliable ML systems that can positively impact our world.

The journey of a thousand lines of code begins with a single API call. As the research community continues to navigate the complex landscape of ML APIs, may this exploration proceed with clarity, precision, and a commitment to excellence in software engineering.

Bibliography

[1] Linkedin. (2023) Api market 2023 to witness robust expansion by 2030. [Online]. Available: https://www.linkedin.com/pulse/api-market-2023-witness-robust-expansion-2030/

[2] J. Ofoeda, R. Boateng, and J. Effah, "Application programming interface (api) research: A review of the past to inform the future," *International Journal of Enterprise Information Systems (IJEIS)*, vol. 15, no. 3, pp. 76–95, 2019.

[3] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. R. Lyu, "Revisiting, benchmarking and exploring api recommendation: How far are we?" *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1876–1897, 2022.

[4] D. Hou and D. M. Pletcher, "Towards a better code completion system by api grouping, filtering, and popularity-based ranking," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 26–30.

[5] Y. Zhou, X. Yang, T. Chen, Z. Huang, X. Ma, and H. Gall, "Boosting api recommendation with implicit feedback," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2157–2172, 2021.

[6] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "Clear: contrastive learning for api recommendation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 376–387.

[7] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.

[8] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 349–359.

[9] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[11] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 207–216.

[12] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected API subgraph via text phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[13] Y. Kang, Z. Wang, H. Zhang, J. Chen, and H. You, "Apirecx: Cross-library api recommendation via pre-trained language model," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3425–3436.

[14] C. Chen, X. Peng, Z. Xing, J. Sun, X. Wang, Y. Zhao, and W. Zhao, "Holistic combination of structural and textual code information for context based api recommendation," *IEEE Transactions on Software Engineering*, 2021.

[15] S. Gao, L. Liu, Y. Liu, H. Liu, and Y. Wang, "Api recommendation for the development of android app features based on the knowledge mined from app stores," *Science of Computer Programming*, vol. 202, p. 102556, 2021.

[16] X. Gu, H. Zhang, and S. Kim, "Codekernel: A graph kernel based approach to the selection of api usage examples," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 590–601.

[17] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Codegru: Context-aware deep learning with gated recurrent unit for source code modeling," *Information and Software Technology*, vol. 125, p. 106309, 2020.

[18] J. Liu, Y. Qiu, Z. Ma, and Z. Wu, "Autoencoder based api recommendation system for android programming," in *2019 14th International Conference on Computer Science & Education (ICCSE)*. IEEE, 2019, pp. 273–277.

[19] J. Yan, Y. Qi, Q. Rao, H. He *et al.*, "Learning api suggestion via single lstm network with deterministic negative sampling." in *SEKE*, 2018, pp. 137–136.

[20] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning api usages from bytecode: A statistical approach," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 416–427.

[21] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 293–304.

[22] J. Zhang, H. Jiang, Z. Ren, and X. Chen, "Recommending apis for api related questions in stack overflow," *IEEE Access*, vol. 6, pp. 6205–6219, 2017.

[23] M. Liu, Y. Yang, Y. Lou, X. Peng, Z. Zhou, X. Du, and T. Yang, "Recommending analogical apis via knowledge graph embedding," *arXiv preprint arXiv:2308.11422*, 2023.

[24] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.

[25] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 290–300.

[26] G. Ajam, B. Hilla, C. Rodrıguez, and B. Benatallah, "Scout-bot: Leveraging api community knowledge for exploration and discovery of api learning resources." *CLEI electronic journal*, vol. 24, no. 2, 2021.

[27] H. Yin, Y. Zheng, Y. Sun, and G. Huang, "An api learning service for inexperienced developers based on api knowledge graph," in *2021 IEEE International Conference on Web Services (ICWS)*. IEEE, 2021, pp. 251–261.

[28] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching documents with examples: A corpus mining approach," *ACM Transactions on Information Systems (TOIS)*, vol. 31, no. 1, pp. 1–27, 2013.

[29] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270.

[30] W. Xie, X. Peng, M. Liu, C. Treude, Z. Xing, X. Zhang, and W. Zhao, "Api method recommendation via explicit matching of functionality verb phrases," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1015–1026.

[31] W. Yuan, H. H. Nguyen, L. Jiang, Y. Chen, J. Zhao, and H. Yu, "Api recommendation for event-driven android application development," *Information and Software Technology*, vol. 107, pp. 30–47, 2019.

[32] J. Sun, Z. Xing, X. Peng, X. Xu, and L. Zhu, "Task-oriented api usage examples prompting powered by programming task knowledge graph," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 448–459.

[33] Y. Zhao, L. Li, X. Sun, P. Liu, and J. Grundy, "Icon2code: Recommending code implementations for android gui components," *Information and Software Technology*, vol. 138, p. 106619, 2021.

[34] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1407–1410.

[35] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 858–868.

[36] C. Ling, Y. Zou, and B. Xie, "Graph neural network based collaborative filtering for api usage recommendation," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 36–47.

[37] J. Jones, "Abstract syntax tree implementation idioms," in *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, 2003, pp. 1–10.

[38] Z. Cai, L. Lu, and S. Qiu, "An abstract syntax tree encoding method for cross-project defect prediction," *IEEE Access*, vol. 7, pp. 170 844–170 853, 2019.

[39] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 359–368.

[40] A. Koyuncu, K. Liu, T. F. Bissyande, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[41] D. Tran and H. Nguyen, "Api specification-based function search engine using natural language query," in *2013 International Conference on Computing, Management and Telecommunications (ComManTel)*. IEEE, 2013, pp. 140–145.

[42] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel, "Identifier-based context-dependent api method recommendation," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 31–40.

[43] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 357–367.

[44] J. Liu and Z. Ma, "Design of an api recommendation system in android programming," in *Journal of Physics: Conference Series*, vol. 1195, no. 1. IOP Publishing, 2019, p. 012003.

[45] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 438–449.

[46] W. K. Lee and M. T. Su, "Mining stack overflow for api class recommendation using doc2vec and lda," *IET Software*, vol. 15, no. 5, pp. 308–322, 2021.

[47] T. Van Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, and T. N. Nguyen, "Combining word2vec with revised vector space model for better code retrieval," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 183–185.

[48] Y. Tian, X. Wang, H. Sun, Y. Zhao, C. Guo, and X. Liu, "Automatically generating api usage patterns from natural language queries," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 59–68.

[49] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[50] Y. Wang, Y. Zhou, T. Chen, J. Zhang, W. Yang, and Z. Huang, "Hybrid collaborative filtering-based api recommendation," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 906–914.

[51] Y. Xie, J. Lin, H. Dong, L. Zhang, and Z. Wu, "Survey of code search based on deep learning," *ACM Transactions on Software Engineering and Methodology*, 2023.

[52] J. Shin and J. Nam, "A survey of automatic code generation from natural language," *Journal of Information Processing Systems*, vol. 17, no. 3, pp. 537–555, 2021.

[53] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[54] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[55] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon, "What programmers really want: results of a needs assessment for sdk documentation," in *Proceedings of the 20th annual international conference on Computer documentation*, 2002, pp. 133–141.

[56] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 international workshop on Mining software repositories*, 2005, pp. 1–5.

[57] C.-Y. Ling, Y.-Z. Zou, Z.-Q. Lin, and B. Xie, "Graph embedding based api graph search and recommendation," *Journal of Computer Science and Technology*, vol. 34, no. 5, pp. 993–1006, 2019.

[58] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM systems journal*, vol. 46, no. 2, pp. 265–288, 2007.

[59] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.

[60] S. Lee, R. Wu, S.-C. Cheung, and S. Kang, "Automatic detection and update suggestion for outdated api names in documentation," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 653–675, 2019.

[61] C. Malamud, *Analyzing Novell Networks*. John Wiley & Sons, Inc., 1991.

[62] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 319–328.

[63] M. Linares-Vasquez, G. Bavota, C. Bernal-Cardenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 2–11.

[64] J. Steele and N. To, *The Android developer's cookbook: building applications with the Android SDK*. Pearson Education, 2010.

[65] Y. Xi, "Api parameter recommendation based on documentation analysis," Master's thesis, University of Waterloo, 2020.

[66] A. R. D'Souza, D. Yang, and C. V. Lopes, "Collective intelligence for smarter api recommendations in python," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 51–60.

[67] M. Monperrus, E. Campos, and M. Maia, "Searching stack overflow for api-usage-related bug fixes using snippet-based queries," in *26th Annual International Conference on Computer Science and Software Engineering*, 2016.

[68] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*. PMLR, 2014, pp. 649–657.

[69] L. Qi, Q. He, F. Chen, X. Zhang, W. Dou, and Q. Ni, "Data-driven web apis recommendation for building web applications," *IEEE transactions on big data*, 2020.

[70] X. Wang, H. Wu, and C.-H. Hsu, "Mashup-oriented api recommendation via random walk on knowledge graph," *IEEE Access*, vol. 7, pp. 7651–7662, 2018.

[71] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending api usages for mobile apps with hidden markov model," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 795–800.

[72] Z. Chen, T. Zhang, and X. Peng, "A novel api recommendation approach by using graph attention network," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 726–737.

[73] X. Liu, L. Huang, and V. Ng, "Effective API recommendation without historical software repositories," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 282–292.

[74] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "A simple, efficient, context-sensitive approach for code completion," *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 512–541, 2016.

[75] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, and B. Xu, "Pyart: Python api recommendation in real-time," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1634–1645.

[76] Z. Wang, S. Huang, Z. Liu, M. Yan, X. Xia, B. Wang, and D. Yang, "Plot2api: recommending graphic api from plot via semantic parsing guided neural network," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 458–469.

[77] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta, "Recommending api function calls and code snippets to support software development," *IEEE Transactions on Software Engineering*, 2021.

[78] D. Fucci, A. Mollaalizadehbahnemiri, and W. Maalej, "On using machine learning to identify knowledge in api reference documentation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 109–119.

[79] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[80] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.

[81] B. Kitchenham, L. Madeyski, and D. Budgen, "Segress: Software engineering guidelines for reporting secondary studies," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1273–1298, 2022.

[82] M. M. Rahman and C. K. Roy, "A systematic review of automated query reformulations in source code search," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–79, 2023.

[83] L. Yang, H. Chen, Z. Li, X. Ding, and X. Wu, "Chatgpt is not enough: Enhancing large language models with knowledge graphs for fact-aware language modeling," *arXiv preprint arXiv:2306.11489*, 2023.

[84] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language model: Survey, landscape, and vision," 2023.

[85] J. Wang, X. Hu, W. Hou, H. Chen, R. Zheng, Y. Wang, L. Yang, H. Huang, W. Ye, X. Geng *et al.*, "On the robustness of chatgpt: An adversarial and out-of-distribution perspective," *arXiv preprint arXiv:2302.12095*, 2023.

[86] CoreRank. (2023) Core ranking system. [Online]. Available: http://portal.core.edu.au/conf-ranks/

[87] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting software developers with a holistic recommender system," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 94–105.

[88] H. Li, T. Li, S. Zhong, Y. Kang, and T. Chen, "A fusion of java domain knowledge base and siamese network for java api recommendation," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2020, pp. 398–405.

[89] Z. T. Sworna, C. Islam, and M. A. Babar, "Apiro: A framework for automated security tools api recommendation," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–42, 2023.

[90] X. Sun, C. Xu, B. Li, Y. Duan, and X. Lu, "Enabling feature location for api method recommendation and usage location," *IEEE Access*, vol. 7, pp. 49 872–49 881, 2019.

[91] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

[92] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical software engineering*, vol. 19, pp. 619–654, 2014.

[93] Mendableai, "Mendableai/firecrawl: turn entire websites into llm-ready markdown or structured data. scrape, crawl and extract with a single api." [Online]. Available: https://github.com/mendableai/firecrawl

[94] LiveBench, "Livebench/livebench: Livebench: A challenging, contamination-free llm benchmark." [Online]. Available: https://github.com/LiveBench/LiveBench

[95] B. A. Campbell and C. Treude, "Nlp2code: Code snippet content assist via natural language tasks," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 628–632.

[96] S.-J. Lee, X. Lin, W.-C. Su, and H.-M. Chen, "A comment-driven approach to api usage patterns discovery and search," *Journal of Internet Technology*, vol. 19, no. 5, pp. 1587–1601, 2018.

[97] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[98] W. Yuan, H. H. Nguyen, L. Jiang, and Y. Chen, "Libraryguru: Api recommendation for android developers," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 364–365.

[99] K. Richardson and J. Kuhn, "Function assistant: A tool for nl querying of apis," *arXiv preprint arXiv:1706.00468*, 2017.

[100] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[101] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2727–2735.

[102] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 404–415.

[103] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Recommending framework extension examples," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 456–466.

[104] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[105] Z. Eberhart and C. McMillan, "Dialogue management for interactive api search," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 274–285.

[106] Z. Ma, S. An, B. Xie, and Z. Lin, "Compositional api recommendation for library-oriented code generation," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 87–98.

[107] Y. Wang, L. Chen, C. Gao, Y. Fang, and Y. Li, "Prompt enhance api recommendation: visualize the user's real intention behind this query," *Automated Software Engineering*, vol. 31, no. 1, p. 27, 2024.

[108] C. Xu, X. Sun, B. Li, X. Lu, and H. Guo, "Mulapi: Improving api method recommendation with api usage location," *Journal of Systems and Software*, vol. 142, pp. 195–205, 2018.

[109] Y. Chen, C. Gao, M. Zhu, Q. Liao, Y. Wang, and G. Xu, "Apigen: Generative api method recommendation," *arXiv preprint arXiv:2401.15843*, 2024.

[110] M. Wei, Y. Huang, J. Wang, J. Shin, N. S. Harzevili, and S. Wang, "Api recommendation for machine learning libraries: how far are we?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 370–381.

[111] J. Liu, B. Liu, W. Dong, Y. Zhang, and D. Wang, "How much support can api recommendation methods provide for component-based synthesis?" in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 872–881.

[112] W. Wen, S. Wang, B. Ye, X. Zhu, Y. Hu, X. Lu, and B. Zhang, "Api recommendation based on wii-wmd," *International Journal of Cognitive Informatics and Natural Intelligence (IJCINI)*, vol. 15, no. 4, pp. 1–20, 2021.

[113] M. Etter and F. Mehta, "Towards type-directed api search for mainstream languages," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development*, 2024, pp. 50–61.

[114] M. M. Rahman and C. Roy, "Nlp2api: Query reformulation for code search using crowdsourced knowledge and extra-large data analytics," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 714–714.

[115] X. Li, L. Liu, Y. Liu, and H. Liu, "A lightweight api recommendation method for app development based on multi-objective evolutionary algorithm," *Science of Computer Programming*, vol. 226, p. 102927, 2023.

[116] T. Nguyen, N. Tran, H. Phan, T. Nguyen, L. Truong, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Complementing global and local contexts in representing api descriptions to improve api retrieval tasks," in *Proceedings of the 2018 26th ACM Joint Meeting*

on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 551–562.

[117] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[118] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[119] I. C. Irsan, T. Zhang, F. Thung, K. Kim, and D. Lo, "Picaso: enhancing api recommendations with relevant stack overflow posts," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 92–37.

[120] Z. Li, C. Li, Z. Tang, W. Huang, J. Ge, B. Luo, V. Ng, T. Wang, Y. Hu, and X. Zhang, "Ptm-apirec: Leveraging pre-trained models of source code in api recommendation," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 3, pp. 1–30, 2024.

[121] I. C. Irsan, T. Zhang, F. Thung, K. Kim, and D. Lo, "Multi-modal api recommendation," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 272–283.

[122] K. Li, X. Tang, F. Li, H. Zhou, C. Ye, and W. Zhang, "Pybartrec: Python api recommendation with semantic information," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 33–43.

[123] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[124] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[125] J. Zhang, Y. Zhao, M. Saleh, and P. Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization," in *International Conference on Machine Learning*. PMLR, 2020, pp. 11 328–11 339.

[126] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer." *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[127] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[128] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[129] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020.

[130] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[131] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.

[132] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[133] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding with unsupervised learning," 2018.

[134] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[135] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[136] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *arXiv preprint arXiv:2305.15334*, 2023.

[137] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, Q. Lin, and D. Jiang, "Wizardlm: Empowering large pre-trained language models to follow complex instructions," in *The Twelfth International Conference on Learning Representations*, 2024.

[138] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017.

[139] M. M. Rahman and C. Roy, "Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 473–484.

[140] R. Silva, C. Roy, M. Rahman, K. Schneider, K. Paixao, and M. Maia, "Recommending comprehensive solutions for programming tasks by mining crowd knowledge," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 358–368.

[141] S. Overflow, https://stackoverflow.com/, 2008.

[142] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[143] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[144] A. v. d. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv preprint arXiv:1807.03748*, 2018.

[145] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[146] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[147] R. Řehřek, P. Sojka *et al.*, "Gensim—statistical semantics in python," *Retrieved from genism. org*, 2011.

[148] X. Rong, "word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[149] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

[150] A. H. Ombabi, O. Lazzez, W. Ouarda, and A. M. Alimi, "Deep learning framework based on word2vec and cnnfor users interests classification," in *2017 Sudan conference on computer science and information technology (SCCSIT)*. IEEE, 2017, pp. 1–7.

[151] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv preprint arXiv:1909.10351*, 2019.

[152] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[153] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," in *International workshop on similarity-based pattern recognition*. Springer, 2015, pp. 84–92.

[154] F. Thabtah, S. Hammoud, F. Kamalov, and A. Gonsalves, "Data imbalance in classification: Experimental evaluation," *Information Sciences*, vol. 513, pp. 429–441, 2020.

[155] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," *arXiv preprint arXiv:2004.11362*, 2020.

[156] J. Snell, K. Swersky, and R. S. Zemel, "Prototypical networks for few-shot learning," *arXiv preprint arXiv:1703.05175*, 2017.

[157] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[158] E. Bisong, "Google colaboratory," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 59–64.

[159] T. Chen, S. Kornblith, K. Swersky, M. Norouzi, and G. Hinton, "Big self-supervised models are strong semi-supervised learners," *arXiv preprint arXiv:2006.10029*, 2020.

[160] E. Voorhees, "Proceedings of the 8th text retrieval conference," *TREC-8 Question Answering Track Report*, pp. 77–82, 1999.

[161] D. R. Radev, H. Qi, H. Wu, and W. Fan, "Evaluating web-based question answering systems." in *LREC*. Citeseer, 2002.

[162] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 39.

[163] B. Everitt and A. Skrondal, *The Cambridge dictionary of statistics*. Cambridge University Press Cambridge, 2002, vol. 106.

[164] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[165] R. J. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 2013, pp. 160–172.

[166] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[167] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 104–115.

[168] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.

[169] G. Nguyen, S. Dlugolinsky, M. Bobak, V. Tran, A. L. Garcia, I. Heredia, P. Malik, and L. Hluchy, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[170] T. Dey, A. Karnauch, and A. Mockus, "Representation of developer expertise in open source software," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 995–1007.

[171] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1548–1560.

[172] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," in *2013 IEEE international conference on big data*. IEEE, 2013, pp. 111–118.

[173] (2021) Python package index - pypi. [Online]. Available: https://pypi.org/

[174] (2021) Pypi download stats. [Online]. Available: https://pypistats.org/

[175] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[176] W. McKinney, *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.", 2012.

[177] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[178] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[179] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, "Tensorflow distributions," *arXiv preprint arXiv:1711.10604*, 2017.

[180] N. Ketkar, "Introduction to keras," in *Deep learning with Python*. Springer, 2017, pp. 97–111.

[181] T. Drabas and D. Lee, *Learning PySpark*. Packt Publishing Ltd, 2017.

[182] "Query stackoverflow - stack exchange data explorer," Available at https://data.stackexchange.com/stackoverflow/query/new (2021/08/01), 2021.

[183] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 127–137.

[184] B. Xu, Z. Xing, X. Xia, D. Lo, and S. Li, "Domain-specific cross-language relevant question retrieval," *Empirical Software Engineering*, vol. 23, no. 2, pp. 1084–1122, 2018.

[185] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: Automated generation of answer summary to developers' technical questions," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 706–716.

[186] X. Xia and D. Lo, "An effective change recommendation approach for supplementary bug fixes," *automated software engineering*, vol. 24, no. 2, pp. 455–498, 2017.

[187] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 345–355.

[188] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 476–481.

[189] B. Vasilescu, V. Filkov, and A. Serebrenik, "Stackoverflow and github: Associations between software development and crowdsourced knowledge," in *2013 International Conference on Social Computing*. IEEE, 2013, pp. 188–195.

[190] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 25–34.

[191] S. Wang, D. Lo, and L. Jiang, "An empirical study on developer interactions in stackoverflow," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1019–1024.

[192] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215. Citeseer, 1994, pp. 487–499.

[193] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 625–635.

[194] S. Dutta, D. Wei, H. Yueksel, P.-Y. Chen, S. Liu, and K. Varshney, "Is there a trade-off between fairness and accuracy? a perspective using mismatched hypothesis testing," in *International Conference on Machine Learning*. PMLR, 2020, pp. 2803–2813.

[195] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 392–403.

[196] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 880–890.

[197] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 869–879.

[198] H. Jiang, J. Zhang, Z. Ren, and T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for apis," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 38–48.

[199] M. I. Razzak, S. Naz, and A. Zaib, "Deep learning for medical image processing: Overview, challenges and the future," *Classification in BioApps: Automation of Decision Making*, pp. 323–350, 2018.

[200] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer, "Defining and substantiating the terms scene, situation, and scenario for automated driving," in *2015 IEEE 18th international conference on intelligent transportation systems*. IEEE, 2015, pp. 982–988.

[201] G. Hu, Y. Yang, D. Yi, J. Kittler, W. Christmas, S. Z. Li, and T. Hospedales, "When face recognition meets with deep learning: an evaluation of convolutional neural networks for face recognition," in *Proceedings of the IEEE international conference on computer vision workshops*, 2015, pp. 142–150.

[202] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, "The rise of deep learning in drug discovery," *Drug discovery today*, vol. 23, no. 6, pp. 1241–1250, 2018.

[203] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 2, pp. 604–624, 2020.

[204] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 886–896.

[205] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: A benchmark for api-misuse detectors," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 464–467.

[206] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static api-misuse detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 265–275.

[207] W. Baker, M. O'Connor, S. R. Shahamiri, and V. Terragni, "Detect, fix, and verify tensorflow api misuses," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 925–929.

[208] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[209] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Are machine learning cloud apis used correctly?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 125–137.

[210] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 129–140.

[211] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[212] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

[213] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 240–250.

[214] C. Lindig, "Mining patterns and violations using concept analysis," in *The Art and Science of Analyzing Software Data*. Elsevier, 2015, pp. 17–38.

[215] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Automated Software Engineering*, vol. 18, pp. 263–292, 2011.

[216] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[217] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[218] N. M. S. Surameery and M. Y. Shakor, "Use chat gpt to solve programming bugs," *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, vol. 3, no. 01, pp. 17–22, 2023.

[219] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *arXiv preprint arXiv:2303.07839*, 2023.

[220] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. Günnemann, E. Hüllermeier *et al.*, "Chatgpt for good? on opportunities and challenges of large language models for education," *Learning and Individual Differences*, vol. 103, p. 102274, 2023.

[221] R. Aiyappa, J. An, H. Kwak, and Y.-Y. Ahn, "Can we trust the evaluation on chatgpt?" *arXiv preprint arXiv:2303.12767*, 2023.

[222] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[223] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[224] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[225] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[226] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv e-prints*, pp. arXiv–1605, 2016.

[227] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 2135–2135.

[228] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, R. Mitchell, I. Cano, T. Zhou *et al.*, "Xgboost: extreme gradient boosting," *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.

[229] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[230] X. Li, J. Jiang, S. Benton, Y. Xiong, and L. Zhang, "A large-scale study on api misuses in the wild," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 241–252.

[231] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[232] B. Glaser and A. Strauss, *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017.

[233] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 802–814.

[234] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," *arXiv preprint arXiv:2111.03922*, 2021.

[235] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *arXiv preprint arXiv:2305.04764*, 2023.

[236] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 483–494.

[237] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 70–79.

[238] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 155–165.

[239] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1027–1038.

[240] G. van Rossum, B. Warsaw, and N. Coghlan, "Style guide for Python code," PEP 8, 2001. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[241] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," *arXiv preprint arXiv:2301.08653*, 2023.

[242] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[243] A. Cheshkov, P. Zadorozhny, and R. Levichev, "Evaluation of chatgpt model for vulnerability detection," *arXiv preprint arXiv:2304.07232*, 2023.

[244] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, "Generating secure hardware using chatgpt resistant to cwes," *Cryptology ePrint Archive*, 2023.

[245] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *arXiv preprint arXiv:2305.01210*, 2023.

[246] Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, "Investigating code generation performance of chat-gpt with crowdsourcing social data," in *Proceedings of the 47th IEEE Computer Software and Applications Conference*, 2023, pp. 1–10.

[247] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by chatgpt?" *arXiv preprint arXiv:2304.09655*, 2023.

[248] T. Zhang, T. Yu, T. Hashimoto, M. Lewis, W.-t. Yih, D. Fried, and S. Wang, "Coder reviewer reranking for code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 41 832–41 846.

[249] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.

[250] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: analysis for machine learning programs," in *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 1–10.

[251] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.