Improving Programming Productivity with Statistical Models

by

Tam Nguyen

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 7, 2022

Keywords: Software engineering, Statistical models, API usages, Exception handling, Personalized models

Approved by

Tung Nguyen, Chair, Assistant Professor of Computer Science and Software Engineering
Cheryl Seals, Co-Chair, Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering
Xiao Qin, Professor of Computer Science and Software Engineering

Abstract

Modern applications often need to have very short time-to-market and upgrade cycles, and thus, short development time. To address this requirement, application developers often rely heavily on API frameworks and libraries when developing apps. However, learning the usages of API methods and objects is often challenging due to the fast-changing nature of API frameworks and the insufficiency of API documentation and source code examples. In our research, we focus on API usage patterns, i.e., frequently occurring patterns that appeared when using API objects and methods. Specifically, we develop statistical models that capture API usage patterns and use those models to improve programming productivity.

This dissertation has three contributions. One is SALAD, a novel approach to address the problem of learning API mobile frameworks. SALAD can learn complex API usages involving several API objects and methods. SALAD learns the usages from bytecode of Android mobile apps, of which millions are publicly available. The main component of SALAD is HAPI, a statistical model of API usages and three algorithms to extract method call sequences from apps' bytecode, to train HAPI based on those sequences, and to recommend method calls using the trained HAPIs. SALAD can automatically generate recommendations for incomplete API usages, thus it could reduce the chance of API usage errors and improve code quality.

The other main contribution of this dissertation is FUZZYCATCH, a code recommendation approach for exception handling. FUZZYCATCH learns usage rules involving API methods, exceptions, and handling actions from a large collection of high-quality apps publicly available. Based on fuzzy logic rules learned from thousands of those apps, SALAD can predict if a runtime exception potentially occurs in a code snippet. Then, as the programmer requests, it can generate the try-catch statement with catch block containing code to catch that exception and the exception handling actions to recover from the exception.

The final contribution is PERSONA, a novel code recommendation model that focuses on the personal coding patterns of programmers while also combines with project-specific and common code patterns. As a personalized model, PERSONA is built and updated for each programmer. It is composed of three sub-models: a model that captures personal code patterns of a programmer; a model that captures the project-level code patterns that the programmer is working on; and a general model that captures code patterns shared between multiple projects. PERSONA incorporates code patterns learned from the three sub-models together and utilizes those patterns for recommending code elements including variable names, class names, methods, and parameters.

Acknowledgments

First of all, I would like to thank my wife, Linh Doan, for everything she has given me during my Ph.D. study. She has always been with me, sharing with me all the joys and sorrows in study and life. She left behind all her great successes, family, and friends in our country to be here with me in the United States. She supported me with everything she has for me to focus on my Ph.D. study, work on this dissertation, and look for jobs. She encouraged me and helped me get through the most difficult moments. My wife and my son, Henry, are everything to me. They are my biggest encouragement to keep me working hard in this long, challenging, but valuable journey.

I would like to dedicate this work to my family, especially, my father - Hai Nguyen, and my uncle - Thanh Nguyen. Due to war and poverty, my father could not get a complete education, thus, he has taken every opportunity for me to get the best education that I could have. I will always remember the day he bought the first computer for me to study programming, the days he drove me from my house to the center of Hanoi for me to study before I joined one of the best high schools in the country. My father was always there for me, giving me advice, and making sure I am on the right track in my life and education. I would not be who I am today without him. I also would like to give a special thank to my uncle, Thanh Nguyen. He lost most of his health fighting for the independence of my country. Coming back from the war, he overcame all the challenges to become a high school teacher. He understands that the only way for our country and our future generation to be better is through education. He taught me mathematics since I was five. His story has transformed me and motivated me to always seek education. Additionally, I would like to thank my grandparents, my parents-in-law, aunts, uncles, brothers, and sisters, who have supported me all the time with great love.

I would like to express my special thanks to my advisor and my teammates, who have helped me get through my Ph.D. study. First and foremost, I would like to thank my supervisor, Dr. Tung Nguyen, for his guidance and support toward my research and career development. His academic journey has inspired me to start my Ph.D. study. He taught and trained me to become a researcher. Under his supervision, I pursued the most exciting research problems and his words of encouragement have kindled motivated me to get through the most difficult time. Without his support, I might not have been succeeded in my Ph.D. study. I also sincerely appreciate my collaborators and teammates: Hung Pham, and Phong Vu, who have studied and worked beside me very closely from the time I started my program. They not only contributed significantly to my research projects but also supported me gracefully in my personal life.

Finally, I would like to thank my committee members, Dr. Cheryl Seals, Dr. Tung Nguyen, Dr. James Cross, Dr. Dean Hendrix, and Dr. Xiao Qin, for their time evaluating my work, attending my exams, and recommending me for future job opportunities. Especially, I am grateful to Dr. Seals for helping me complete this dissertation and preparing for the exams. I also would like to thank Dr. SueAnne Nichole Griffith for her approval to serve as the university reader and to evaluate this dissertation. Last but not least, I would like to thank The Department of Computer Science and Software Engineering at Auburn University, and The Computer Science Department at Utah State University for their financial support for my Ph.D. study.

Publications

14. **Tam Nguyen**, Tung Nguyen (2021). PERSONA: A personalized model for code recommendation. *PLoS ONE 16(11): e0259834*. https://doi.org/10.1371/journal.pone.0259834

13. **Tam Nguyen**, Phong Vu, Tung Nguyen (2020). Code Recommendation for Exception Handling. *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. (ESEC/FSE 2020). https://doi.org/10.1145/3368089.3409690

12. **Tam Nguyen**, Phong Vu, Tung Nguyen (2020). API Misuse Correction: A Fuzzy Logic Approach. *The 2020 ACM Southeast Conference*. (ACMSE 2020). https://doi.org/10.1145/3374135.3385307

11. **Tam Nguyen**, Phong Vu, Tung Nguyen (2019). Personalized Code Recommendation. *The 35^{th} International Conference on Software Maintenance and Evolution*. (ICSME 2019). https://doi.org/DOI:10.1109/ICSME.2019.00047

10. **Tam Nguyen**, Phong Vu, Tung Nguyen (2019). An empirical study of exception handling bugs and fixes. *The 2019 ACM Southeast Conference*. (ACMSE 2019). https://doi.org/10.1145/3299815.3314472

9. **Tam Nguyen**, Phong Vu, Tung Nguyen (2019). Code Search on Bytecode for Mobile App Development. *The 2019 ACM Southeast Conference*. (ACMSE 2019). https://doi.org/10.1145/3299815.3314471

8. **Tam Nguyen**, Phong Vu, Hung Pham, Tung Nguyen (2018). Deep Learning UI Design Patterns of Mobile Apps. *The* 40^{th} *International Conference on Software Engineering.* (ICSE 2018). `https://doi.org/10.1145/3183399.3183422`

7. **Tam Nguyen**, Phong Vu, Hung Pham, Tung Nguyen (2018). Recommending Exception Handling Patterns with ExAssist. *The* 40^{th} *International Conference on Software Engineering.* (ICSE 2018). `https://doi.org/10.1145/3183440.3194971`

6. Phong Vu, **Tam Nguyen**, Hung Pham, Tung Nguyen (2018). Advanced Linguistic Pattern and Concept Analysis Framework for Software Engineering Corpora. *The* 40^{th} *International Conference on Software Engineering.* (ICSE 2018). `https://doi.org/10.1145/3183440.3194972`

5. **Tam Nguyen**, Hung Pham, Phong Vu, Tung Nguyen (2016). Learning API Usages from Bytecode: A Statistical Approach. *The* 38^{th} *International Conference on Software Engineering.* (ICSE 2016). `https://doi.org/10.1145/2884781.2884873`

4. Phong Vu, Hung Pham, **Tam Nguyen**, Tung Nguyen (2016). Phrase-Based Extraction of User Opinions in Mobile App Reviews. *The* 31^{st} *IEEE/ACM International Conference on Automated Software Engineering.* (ASE 2016). `https://doi.org/10.1145/2970276.2970365`

3. **Tam Nguyen**, Hung Pham, Phong Vu, Tung Nguyen (2015). Recommending API Usages for Mobile Apps with Hidden Markov Models. *The* 30^{th} *IEEE/ACM International Conference on Automated Software Engineering.* (ASE 2015). `https://doi.org/10.1109/ASE.2015.109`

2. Phong Vu, **Tam Nguyen**, Hung Pham, Tung Nguyen (2015). Mining User Opinions in Mobile App Reviews: A Keyword-based Approach. *The* 30^{th} *IEEE/ACM International Conference on Automated Software Engineering.* (ASE 2015). `https://doi.org/10.1109/ASE.2015.109`

1. Phong Vu, Hung Pham, **Tam Nguyen**, Tung Nguyen (2015). Tool Support for Analyzing Mobile App Reviews. *The* 30^{th} *IEEE/ACM International Conference on Automated Software Engineering.* (ASE 2015). `https://doi.org/10.1109/ASE.2015.101`

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems and services are increasingly important, involving and improving the work and lives of billions of people. With the huge demand, modern applications often need to have very short time-to-market and upgrade cycles, and thus, short development time. To address this requirement, application developers often rely heavily on API frameworks and libraries such as Android and iOS frameworks, Java APIs, etc and new applications extensively re-use API components.

However, learning to use API objects and methods is usually challenging for developers due to several reasons. First, an API framework often consists of a large number of API methods and objects. Moreover, common API usage scenarios often involve several API elements and follow special rules for pre and post-conditions or control and data flows. The API documentation of such usages is generally insufficient. Thirdly, popular API libraries are often upgraded quickly and include very large changes. Thus, it is difficult for developers to keep up with all the changes and new APIs.

In our research, we focus on API usage patterns, i.e., frequently occurring patterns that appeared when using API objects and methods. Specially, we develop statistical models that capture API usage patterns and use those models to improve programming productivity.

One is Salad [82] ("Statistical Approach for Learning APIs from DVM bytecode"), a novel approach to address the problem of learning API mobile frameworks. To address the problem of insufficient documentation and source code examples, Salad learns API usages from bytecode of Android mobile apps, of which millions are publicly available. As a statistical approach, Salad can learn complex API usages involving several API objects and

methods. Finally, SALAD can automatically generate recommendations for incomplete API usages, thus it could reduce the chance of API usage errors and improve code quality.

Our core contributions include HAPI, a statistical model of API usages and three algorithms to extract method call sequences from apps' bytecode, to train HAPI based on those sequences, and to recommend method calls in code completion using the trained HAPIs. Our empirical evaluation shows that our prototype tool can effectively learn API usages from 200 thousand apps containing 350 million method sequences. It recommends next method calls with top-3 accuracy of 90% and outperforms baseline approaches on average 10-20%.

Based on SALAD, we have developed DroidAssist [79], a recommendation tool for API usages of Android mobile apps. DroidAssist is released as a plugin of Android Studio, the standard IDE for Android apps development. It is incorporated with Android Studio and users can invoke it directly from the current editing view (for method call recommendation) or via the menu (for method sequence validation). The details about our approach and the tool could be found at [82, 79].

Secondly, we introduced FUZZYCATCH [75], a code recommendation approach for exception handling. As a statistical approach, FUZZYCATCH learns usage rules involving API methods, exceptions, and handling actions from a large collection of high-quality programs publicly available. Based on learned rules, FUZZYCATCH can predict if a runtime exception would occur in a given code snippet and recommend code to handle that exception. Additionally, FUZZYCATCH could be used in detecting and fixing real exception bugs.

The empirical evaluation suggests that FUZZYCATCH is highly effective. For example, it has top-1 accuracy of 77% on recommending what exception to catch in a try-catch block and of 70% on recommending what method should be called when such an exception occurs. FUZZYCATCH also achieves a high level of accuracy and outperforms baselines significantly on detecting and fixing real exception bugs.

Based on FUZZYCATCH, we have developed ExAssist [74], a code recommendation tool for exception handling. Similar to DroidAssist, ExAssist is released as a plugin of Android

Studio. ExAssist predicts what types of exceptions could occur in a given piece of code and recommends proper exception handling code for such exceptions. The details about our approach and the tool could be found at [75, 74, 84].

Finally, we proposed PERSONA, a lightweight code recommendation model that focuses on the personal coding patterns of programmers. PERSONA is built and updated for each programmer. To learn personal coding patterns, it utilizes fuzzy logic to model correlation/association between code elements in the code history written by the programmer. PERSONA also incorporates project-specific and common code patterns efficiently to further improve the recommendation accuracy.

We implemented a robust code recommendation system based on PERSONA. The system includes a module to extract the usages of variables, methods, classes, parameters from the code history of a programmer, as well as from a large codebase. The system is designed to train PERSONA efficiently. Furthermore, it also allows PERSONA to be re-trained easily to update the coding preferences of programmers as more training data becomes available.

We performed an extensive evaluation that shows the effectiveness of the approach in code recommendation. PERSONA is trained on a dataset containing 14,807 Java projects, with over 350 million lines. We evaluated the model on 10 big Java projects with the number of commits in each project is ranging from 23,000 to over 400,000. The evaluation results show that PERSONA could achieve high accuracy in code recommendation and outperforms the baselines significantly. We also showed that the model could be re-trained and improves the recommendation accuracy over time as more code of the programmer is available. The details about our approach could be found at [78, 72]

Chapter 2

Literature Review

2.1 Statistical modeling for code

Statistical models for capturing rules and patterns in source code become a hot research topic in software engineering in the recent years. Hassan et al. [36] indicated "natural" software analytics based on statistical modeling will become one of the most important of aspects of software analytics. Hindle et al. [39] shows that source code is repetitive and predictable like natural language, and they adopted n-gram model on lexical tokens to suggest the next token. SLAMC [76] represents code by semantic tokens, i.e., annotations of data types, method/field signatures, etc. rather than lexical tokens. SLAMC combines n-gram modeling of consecutive semantic tokens, topic modeling of the whole code corpus, and bi-gram of related API functions. Tu et al. [104] exploited the localness of source code. White et al. [110] proposed deep learning approach modeling source code. Allamanis and Sutton [5] trains n-gram language model a giga-token source code corpus. NATURALIZE [3] use n-gram language model to learns the style of a codebase and suggest natural identifier names and formatting conventions. Jacob et al. [47] uses n-gram model to learn code templates. Hidden Markov Model has been used infer the next token from user-provided abbreviations [35] and detect coded information islands, such as source code, stack traces, and patches, from free text [20]. Maddison et al. [61] proposed tree-based generative models for source code. Hsiao et al. [42] learns n-gram language model on program dependence graph and uses the model for finding plagiarized code pairs.

Fuzzy-based approaches have been proposed to solve problems in software engineering, such as bug triaging problem [101, 102], automatic tagging [2], bug categorization [22]. Overall, the approaches focus on modeling textual software artifacts.

2.2 API usage patterns

There exist several works that proposed statistical models for learning API usages. SLANG uses n-gram and recurrent neural networks (RNN) to learn API usage patterns per object which are used to predict and suggest next API calls. Nguyen et al. [71] introduced GraLan, a graph-based statistical language model that learns common API usage (sub)graphs from a source code corpus and computes the probabilities of generating new usage graphs given the observed (sub)graphs. Although graphs are better than sequences in capturing context information, the number of sub-graphs can grow exponentially. That means, training sequence-based models would be more time- and space-efficient.

Pattern mining approaches represent usage patterns using various data structure such as sequences, sets, trees, and graphs. JADET [107] extracted a usage model as a set of partial order pairs of method calls. MAPO [116] mined frequent API call sequences and suggests associated code examples. Wang et al. [105] mines succinct and high-coverage API usage patterns from source code. Acharya et al. [1] proposed an approach to mine partial orders among APIs. Buse and Weimer [18] propose an automatic technique for mining succinct and representative human-readable API examples. Other techniques include mining associate rules [60], item sets [16], subgraphs [77], [21], code idioms [6], topic modeling [68], etc.

2.3 Code recommendation

One application of usage patterns mined from existing code is to support code completion. Grapacc [70] mines and stores API usage patterns as graphs and suggest these graphs in current editing code. Bruch et al. proposed three approaches for code completion. First, FreqCCS recommends the most frequently used method call. Second, ArCCS mines associate rules and suggest methods that often occur together. Finally, a best matching neighbors code completion technique that makes used k-nearest-neighbor algorithm. SLANG [92] uses n-gram to suggest the next API call based on a window of $n-1$ previous methods. Precise

[113] mines existing code bases and builds a parameter usage database. Upon request, it queries the database and recommends API parameters. Graphite [88] allows library developers to introduce interactive and highly-specialized code generation interfaces that could interact with users and generates appropriate source code.

Other approaches have been proposed to improve code completion tasks. Robbes et al. [93] improves code completion with program history. They measure the accuracy of replaying entire change history of programs with completion engine and gather information for improvement. In [41], the authors found that ranking method calls by frequency of past usages is effective and propose new strategies for organizing APIs in the code completion proposals. Hill and Rideout [38] match the code fragment under editing with small similar-structure code segments that frequently exist in large software projects. The authors of [63] and [99] use API documentation to suggest source code examples to developers. Holmes and Murphy [40] describe an approach for locating relevant code examples based on heuristically matching with the structure of the code under editing.

2.4 Exception handling

Exception handling recommendation has been studied in several research efforts [91, 14, 12, 15, 66, 32, 67]. Barbosa *et al.* [14] proposed a set of three heuristic strategies used to recommend exception handling code. Rahman *et al.* [91] proposed a context-aware approach that recommends exception handling code examples from a number of GitHub projects. Filho *et al.* [32] proposed ArCatch, an architectural conformance checking solution to deal with the exception handling design erosion. Lie et al. [59] proposed an approach, named EXPSOL, which recommends online threads as solutions for a newly reported exception-related bug. Kistner et al. [54] built a tool that reveals possible sources of exceptions in the code. This tool also provides project-specific recommendations and detects common bad exception handling practices. Montenegro et al. [67] proposed an "exception policy expert" tool which alerts developers about policy violations and can suggest possible handlers for the exceptions. Li

et al. [56] devised a method that automatically recommends exception handling strategies, based on program context.

There exist several methods for mining exception-handling rules. WN-miner [109] and CAR-miner [103] are approaches that use association mining techniques to mine association rules between method calls of `try` and `catch` blocks in exception handling code. Both models are used to detect bugs related to exceptions. Zhong *et al.* [114] proposed an approach named MiMo, that mines repair models for exception-related bugs. Approaches based on association rule mining mines only most frequent rules, i.e. rules that satisfy certain support and confidence. Thus, rules involving rare exceptions or methods with lower support and confidence are undiscoverable.

Exception handling has been studied extensively in software engineering research [95, 30, 85, 25, 23, 13, 87, 62, 28, 27, 49, 50, 51, 17, 26, 64]. We mention here several notable studies. Sena *et al.* [95] presents an empirical study to investigate the exception handling strategies adopted by Java libraries. Ebert *et al.* [30] presented an exploratory study on exception handling bugs by surveying of 154 developers and an analysis of 220 exception handling bugs from two Java programs, Eclipse and Tomcat. Similarly, Nguyen *et al.* [85] performed an empirical study on nearly 300 exception-related bugs and fixes from 10 mobile apps. Coelho *et al.* [25] performed a detailed empirical study on exception-related issues of over 6,000 Java exception stack traces extracted from over 600 open-source Android projects. Padua *et al.* [27] studied exception handling practices with exception flow analysis. Kechagia *et al.* [49] investigated the exception handling mechanisms of the Android APIs.

There are several research efforts focus on the global impact of exceptions [94, 33, 19, 96]. Robillard *et al.* [94] studied the current Java exception handling mechanism. They argued that exceptions are a global design problem, and exceptions source are difficult to predict in advance. Cacho *et al.* [19] presented an aspect-oriented model for exception handling.

2.5 Personalized models

In general, personalized models have been studied extensively in the fields of recommender systems [24, 46] and collaborative filtering [98, 97]. For example, Hwang *et al.* [46] proposed a new recommender system, which employs a genetic algorithm to learn personal preferences of customers and provide tailored suggestions. In software engineering, several personalized approached has been proposed. Jiang *et al.* [48] developed a separate prediction model for each developer to predict software defects. In [31], the author proposed a personalized defect prediction framework that gives instant feedback to the developer at change level, based on historical defect and change data. Wang *et al.* [106] proposed a context-aware personalized task recommendation approach to aid dynamic worker decision in crowd-testing tasks.

Chapter 3

Statistical Approach for Learning API Usages

3.1 Introduction

Mobile apps are software applications developed specially for mobile devices like smartphones and tablets. Due to the exploding in popularity and usage of those devices, millions of mobile apps have been developed and made available to end-users. Due to the fierce competition, those mobile apps often need to have very short time-to-market and upgrade cycles, and thus, short development time. To address this requirement, mobile app developers often rely heavily on API application frameworks and libraries such as Android and iOS frameworks, Java APIs, etc and mobile apps extensively re-use API components. For example, a prior study reports some Android apps having up to 42% of their external dependencies to Android APIs and 68% to Java APIs [100].

Learning API usages is usually challenging due to several reasons. First, an API framework often consists of a large number of components. For example, the Android application framework contains over 3,400 classes and 35,000 methods which are organized in more than 250 packages [57]. Moreover, while common API usage scenarios often involve several API elements and follow special rules for pre- and post-conditions or for control and data flows [77, 70, 29], documentation of such usages is generally insufficient. For example, the Javadoc of a class mainly contains only descriptions of its fields and methods and rarely has code examples describing in detail the usages of its objects and methods [52]. Descriptions and code examples for API usages involving several objects are usually non-existed.

These challenges are even more severe for learning APIs of mobile frameworks. Due to the fast development of mobile devices and the strong competition between software and hardware vendors, those frameworks are often upgraded quickly and include very large

changes. For example, 17 major versions of Android framework containing nearly 100,000 method-level changes have been released within five years [57]. Additionally, source code of most mobile apps is not publicly available. With few apps with source code available, finding and learning code examples from existing mobile app projects would be difficult and insufficient.

We introduce SALAD (*"Statistical Approach for Learning APIs from DVM bytecode"*), a novel approach to address the problem of learning API mobile frameworks. To address the problem of insufficient documentation and source code examples, SALAD learns API usages from *bytecode* of Android mobile apps, of which millions are publicly available. As a statistical approach, SALAD can learn complex API usages involving several API objects and methods. Finally, SALAD can automatically generate recommendations for incomplete API usages, thus it could reduce the chance of API usage errors and improve code quality.

The key component of our approach is HAPI, standing for *"Hidden Markov Model of API usages"*. A HAPI is a Hidden Markov Model (HMM) [90] representing method call sequences involving one or multiple related API objects. It has several states aimed to represent the internal states of the represented API objects. It also associates with probabilities for selecting the starting state for transiting from one state to another, and for calling a method at a given state. Those probabilities describe the statistical patterns of API states and method calls.

SALAD also consists of three new algorithms involving HAPI. One algorithm is designated to train a HAPI i.e. inferring its internal states and estimating the associated probabilities from a very large collection of method call sequences. Another algorithm uses a trained HAPI to compute the generating probabilities of several method sequences and rank those sequences based on those probabilities. This ranking result is used for API usage recommendation.

The last algorithm extracts API method sequences from apps' bytecode which are then used for training HAPIs. It first analyzes bytecode and builds GROUMs. A GROUM (*"GRaph-based Object Uasge Model"*) [77] is a graph-based model in which the nodes represent method calls and data objects and the edges indicate control and data flows between those

methods and objects. With that design, a GROUM can represent an usage scenario including many objects and methods and complex control and data flows between them. Once a GROUM is built, our algorithm can travel all its paths following the control and data dependencies and extracts the corresponding method sequences.

We have conducted several experiments to evaluate the usefulness and effectiveness of SALAD. The experiment results show several important points. First, our approach SALAD is highly efficient and scalable. It is able to extract nearly 350 million method sequences and train 24 thousands HAPIs from more than 200 thousands mobile apps downloaded from Google's Android app market. With that huge amount of training data, SALAD can predict and recommend the next method call for a given API method call sequence with a top-3 accuracy of nearly 90% and a top-10 accuracy of more than 98%. In addition, our model HAPI consistently outperforms two baseline models n-gram and recurrent neural network [92] with the average improvement around 10-20%.

Based on SALAD, we have developed an API usage recommendation tool named DroidAssist. This tool is implemented as a plug-in of Android Studio and available online at `http://useal.cs.usu.edu/droidassist`. More information about this tool can be found in [80] and that website. Source code and experiment data of this work (e.g., extracted API method sequences, GROUMs, and HAPIs) are also made available there.

3.2 Motivation examples

This section briefly introduces API usages and HAPI via an example. Figure 3.1, reproduced from Android Developer website [43], illustrates usages of a MediaRecorder object in Android API framework as a state diagram. This state diagram is a finite state machine in which each node (drawn as a rounded rectangle) represents an internal state of the MediaRecorder object and each edge (drawn as an arrow) represents a state transition when a method (drawn as the label of the edge) is called.

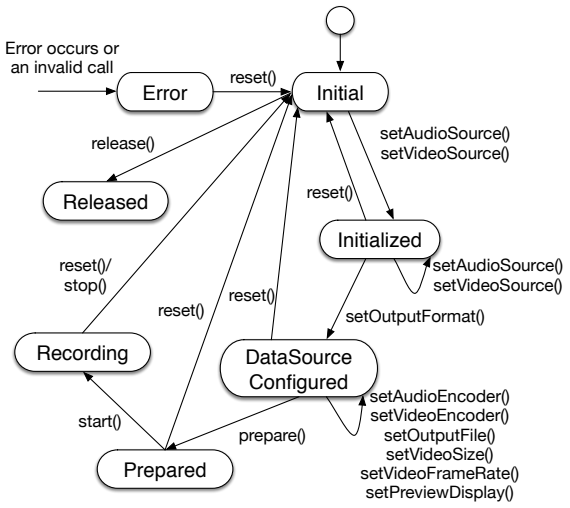

Figure 3.1: State diagram of MediaRecorder object

We learn from this state diagram that a MediaRecorder object has seven states during its lifetime. It is at one state at a time and can change to another state if a method is called. For example, as after being created, the MediaRecorder object is in the *Initial* state. If method setAudioSource or setVideoSource is called, it changes to the *Initialized* state. Then, it will change from the *Initialized* state to the *DataSourceConfigured* state if we call method setOutputFormat. However, at any time, if method reset is called, the object will come back to its *Initial* state.

It could be seen from this example that state diagrams can represent usages of API objects precisely and concisely. Thus, they are useful to learn and understand API usages. For example, we could infer from Figure 3.1 the following method call sequence to perform an audio recording task.

```
setAudioSource(...)
setOutputFormat(...)
setAudioEncoder(...)
setOutputFile(...)
prepare()
start()
stop()
```

```
release()
```

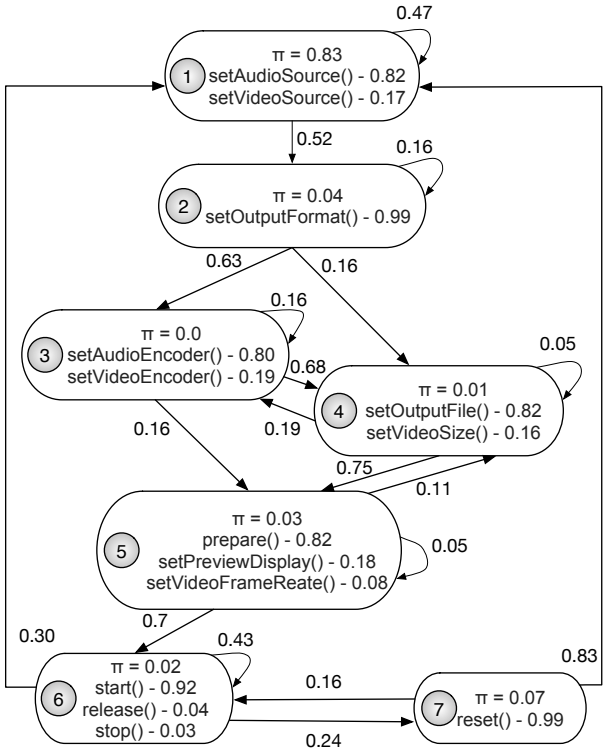

Figure 3.2: HAPI model for usages of MediaRecorder object

Unfortunately, documentation of most of API objects does not contain state diagrams representing their usages. In addition, state diagrams can be used to to check for the *validity* of method call sequences, but not their *popularity*. For example, we cannot know from a state diagram that a sequence is *valid* but *has never been used*, or compare if a sequence *is more likely to be used* in practice than another. Last but not least, current code completion engines like the built-in engine in Android Studio IDE do not incorporate state diagrams in their recommendations.

We design HAPI to take advantages of the usefulness of state diagrams in representing API usages and address their discussed limitations. In essential, a HAPI is a *probabilistic state diagram*. Its nodes and edges associate with probabilities which can be used to estimate the probability of any method call sequences. More importantly, its structure and probabilistic parameters can be learned directly from data.

Figure 3.2 illustrates a HAPI model learned by SALAD to represent the usages of a MediaRecorder object. In this figure, each node of the HAPI represents a state of the object. But different from the state diagram in Figure 3.1, each state of a HAPI has a probability π for being the starting state in a method sequence. It also has a distribution specifying the probability to call a particular method (i.e. emission probabilities) and another distribution specifying the probability to change to another state (i.e. transition probabilities). To simplify the drawing, we only show method calls and transition edges with significant probabilities in the figure. By design, the states of a HAPI model aim to represent the internal states of the corresponding API objects. The transition probabilities capture switching patterns of those internal states; and the emission probabilities describe the method calling patterns at each of those states.

For example, accordingly to the HAPI model in Figure 3.2, a MediaRecorder object starts in state (1) with probability of 83%. In this state, method setAudioSource can be called with a probability of 82% and when that happens, the object can change to state (2) with a probability of 52%. In state (2), setOutputFormat can be called with a probability of 99% and so on. Technically, a HAPI is a Hidden Markov Model [90] which can be trained, i.e. inferring its structure and the associating probabilities, from a given collection of method call sequences. Once trained, we can use it to compute and compare the probabilities of any given method sequences. For example, the sequence setAudioSource, setOutputFormat, setAudioEncoder would have higher probability than the sequence setAudioSource, setOutputFormat, start.

HAPI models have several advantages over plain state diagrams. First, the state diagram only specifies the general rules for using objects but not specify the common usages. In contrast, once learned from code examples, HAPI could infer common usages of an API object by searching for the method sequences that have highest probabilities. Second, a HAPI could be used to predict the most likely next method call in a given API method call sequence. Thus, we can use HAPI to recommend API usages while developers are writing code.

3.3 API usage model

In this section, we will discuss HAPI, our proposed statistical model for API usages in more details. We also re-introduce GROUM, a graph-based model for object usages developed previously by Nguyen *et al.* [77]. SALAD uses GROUM as a temporary representation for API usages extracted from mobile apps' bytecode.

3.3.1 HMM-based API usage model

A HAPI is a generative, probabilistic model that describes the process of generating method call sequences involving one or multiple API objects. As a Hidden Markov Model [90], a HAPI formally has a set $Q = \{q_1, q_2, ..., q_K\}$ of K hidden states and associates with a set $V = \{v_1, v_2, ..., v_M\}$ of M API methods of the API object(s) it is modeling. Each state q_i has a probability π_i to be selected as the starting state of the model. When being in one state, a HAPI can emit (i.e. generate) a method call and/or switch to another state. The transition matrix $A = \{a_{ij} | i, j \in 1..K\}$ specifies the state transition probabilities. That means, a_{ij} is the probability that the model changes from state q_i to state q_j. The generating matrix $B = \{b_{ik} | i \in 1..K, k \in 1..M\}$ specifies the emission probabilities. Specially, b_{ik} is the probability to call method v_k when the model is in state q_i. As seen, this HAPI model has $K + K^2 + KM$ parameters.

With those parameters, the HAPI model can generate a sequence $Y = (y_1, y_2, .., y_T)$ via the following generating process:

1. Set $t = 1$ and sample the state q_{i_t} from the initial state distribution $\pi = \{\pi_1, \pi_2, ..., \pi_K\}$
2. Sample a method call y_t from the calling distribution of state q_{i_t}, i.e. $\{b_{i_t1}, b_{i_t2}, ..., b_{i_tM}\}$
3. Sample the next state $q_{i_{t+1}}$ from the distribution of state q_{i_t}, i.e. $\{a_{i_t1}, a_{i_t2}, ..., a_{i_tK}\}$
4. Increase t and loop back step 2 for T iterations.

Modeling API usages using HAPI involves two problems. The first problem is training (i.e. estimate the parameters of) a HAPI model for one or multiple API objects from existing method call sequences involving. The second problem is using the trained HAPI models to

```
protected void startRecording(String file) {
 MediaRecorder recorder = new MediaRecorder();
 recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
 recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
 recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
 recorder.setOutputFile(file);
 try {
   recorder.prepare();
 } catch (IllegalStateException e) {
   e.printStackTrace();
 } catch (IOException e) {
   e.printStackTrace();
 }
 recorder.start();
}
```

Figure 3.3: Code example using MediaRecorder object

compute the generating probabilities of any method call sequences for recommending API usages. In Section 4, we will present two algorithms we developed in SALAD to solve those two problems.

3.3.2 Graph-based object usage model

A HAPI model for one or multiple API objects is trained by a collection of method sequences involving those objects. To extract those sequences from bytecode, we employ GROUM, a graph-based model of object usages [77], as a temporary representation of API usages.

Figure 3.3 lists a code example with the usage of a MediaRecorder object. Figure 3.4 illustrates the GROUM built for the main execution path of that code example. As seen in this figure, a GROUM is a labeled, directed graph. It has two kinds of nodes: *object nodes* and *action nodes*. An object node represents an object. It is labeled by the name of the object type (e.g. MediaRecorder) and illustrated as a rounded rectangle in the figure. An action node represents a method call. It is labeled the method qualified name (e.g. MediaRecorder.start) and illustrated as a flat rectangle. There are two kinds of edges representing *control flow* and

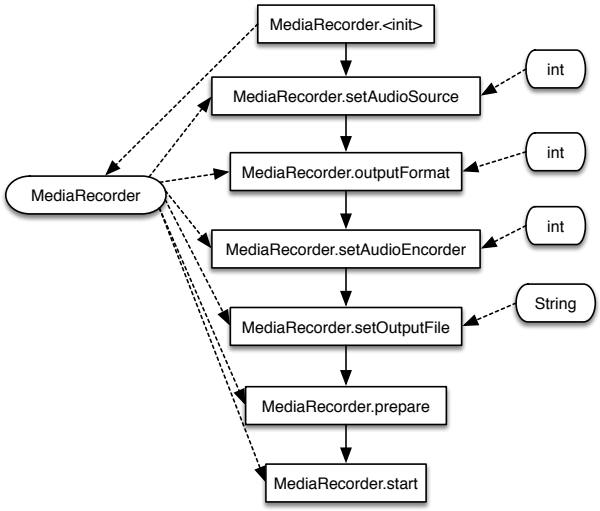

Figure 3.4: GROUM model for usages of MediaRecorder object

data flow. In the figure, solid arrows illustrate control flow edges between action nodes while dashed arrows illustrate data flow edges between object nodes and other nodes.

Using GROUM to represent API usage scenarios has two main advantages. First, we can easily identify all action nodes having data-dependency with a given object node or set of object nodes, from which we can extract method sequences. Second, we could determine if a given set of object nodes have usage-dependency. A set of object nodes has usage-dependency if there is at least one action node that have data-dependency with all of them. SALAD only extracts method sequences of usage-dependent objects because they are more likely to represent legitimate API usages involving multiple objects.

3.4 System architecture

Figure 3.5 shows the overall architecture and processing pipeline of SALAD. For the pre-processing phase, it has two components to extract GROUMs from bytecode and source code and another component to extract API method sequences from those GROUMs. In the training phase, the HAPI Learner component is responsible to use those method sequences to train the HAPI models. Once trained, those models can use two components Method Call

Recommender and Method Sequence Analyzer to recommend a next method calls for a given method sequence and to estimate the likelihood of that sequence, respectively. In the rest of this section, we will present the design and implementation details of those components.

3.4.1 Bytecode GROUM extractor

This component is responsible to extract GROUMs from apps' bytecode instructions. Bytecode of Android apps is stored as .dex files and available on online app markets like Google Play Store. Thus, the extraction process has three steps: 1) Downloading the .dex files, 2) Parsing those files into control flow graphs (CFGs), and 3) Analyzing those CFGs to build the GROUMs.

Downloading .dex files

SALAD has a built-in app crawler developed based on the Google Play Crawler project [45]. This crawler simulates an Android device, thus it can access the app store like a normal smartphone. We program it to download all the top free/new free apps in all categories. For each download application package (.apk files), we extract and store the .dex files which contain all of its bytecode.

Parsing .dex files

The next step is to parse .dex files to build control flow graphs. SALAD employs smali [44], an assembler/disassembler for Android .dex files, to obtain bytecode instructions implemented for each class and method available in the .dex files. Based on those instructions, SALAD constructs a Control Flow Graph (CFG) for each method to simplify the further analysis tasks. Let us discuss this in more details.

Dalvik virtual machine. Android apps are originally developed in Java and re-targeted for execution in Dalvik Virtual Machine (DVM). DVM is a register-based virtual machine. It has a set of 32-bit registers for storing primitive values (e.g. integers and floating point

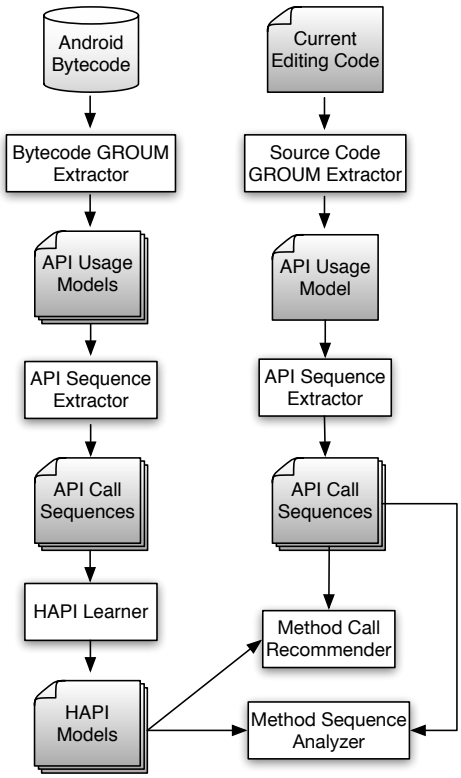

Figure 3.5: The overview of SALAD's architecture

```
public String readTextFile(String fileName) throws IOException {
        FileReader fr = new FileReader(fileName);
        BufferedReader br = new BufferedReader(fr);
        StringBuilder strBuilder = new StringBuilder();
        String line;
        while((line = br.readLine()) != null) {
                strBuilder.append(line);
        }
        br.close();
        return strBuilder.toString();
}
```

Figure 3.6: A source code example

```
|[0001dc] IOManager.readTextFile:(Ljava/lang/String;)Ljava/lang/String;
|0000: new—instance v1, Ljava/io/FileReader;
|0002: invoke—direct {v1, v6}, Ljava/io/FileReader;.<init>:(Ljava/lang/String;)V
|0005: new—instance v0, Ljava/io/BufferedReader;
|0007: invoke—direct {v0, v1}, Ljava/io/BufferedReader;.<init>:(Ljava/io/Reader;)V
|000a: new—instance v3, Ljava/lang/StringBuilder;
|000c: invoke—direct {v3}, Ljava/lang/StringBuilder;.<init>:()V
|000f: invoke—virtual {v0}, Ljava/io/BufferedReader;.readLine:()Ljava/lang/String;
|0012: move—result—object v2
|0013: if—nez v2, 001d
|0015: invoke—virtual {v0}, Ljava/io/BufferedReader;.close:()V
|0018: invoke—virtual {v3}, Ljava/lang/StringBuilder;.toString:()Ljava/lang/String;
|001b: move—result—object v4
|001c: return—object v4
|001d: invoke—virtual {v3, v2}, Ljava/lang/StringBuilder;.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
|0020: goto 000f
```

Figure 3.7: Dalvik bytecode of code example in Figure 3.6

numbers) and object references. A frame (activation record) of a method has a fixed size and consists of a particular number of registers for storing local variables, parameters (including the this reference), return values, and temporary values.

Figure 3.6 lists a code snippet and Figure 3.7 shows the Dalvik bytecode compiled for the method readTextFile in that snippet. For execution, this method is allocated 7 registers v0-v6. v5 is for the receiver object (i.e. this). v6 is for the parameter, a String object for the file name. Registers v0-v4 are used for local or temporary variables. Instructions in Dalvik bytecode operate on registers. For example, instruction mul-int v2,v5,v3 multiples the values of registers v2 and v5 and stores the result to v3. Or instruction new-instance v1, Ljava/io/FileReader; creates a new FileReader object and returns the reference of that object to register v1.

Control flow graph. We construct Control Flow Graph (CFG) as a representation of bytecode instructions. A node in the constructed CFG contains a single bytecode instruction

```
1  function BuildGROUM(Method M)
2    CFG = BuildCFG(M)
3    A = ∅ //list of GROUMs
4    S_t = CreateStartState(CFG, M)
5    F = ∅
6    Push(F, S_t)
7    while F ≠ ∅
8      S = Pop(F)
9      SN = GetStartNode(S)
10     while SN is not a control node
11       BuildTemporaryGROUM(S, SN)
12       AddExploredNode(S, SN)
13       CN = GetNextNode(SN)
14     if SN is the return node
15       AddGROUM(A, S)
16     else
17       for NN ∈ GetNextNodes(SN)
18         if NN ∉ GetExploredNodes(S)
19           S_c = GetCopy(S)
20           SetStartNode(S_c, NN)
21           Push(F, S_c)
22   return A
```

Figure 3.8: Building GROUM algorithm

and an edge is the control flow between the two instructions. There are two types of nodes in CFG. Control nodes represent control instructions in bytecode, e.g. if, return, throw, or goto instructions. Other instructions are normal nodes. Normal nodes in a CFG only have one out-going edge points to the next instruction while control nodes could have several out-going edges (if, switch nodes) or do not have any (return nodes). Techniques used for constructing CFGs are quite standard. First, we create all nodes in the CFG, each one corresponds to an instruction in the instruction list. Then, for each node, we use offsets to identify nodes that are executed after it and add edges from the current node to those nodes.

Building GROUM

In this step, SALAD takes a CFG as input and produce GROUMs which describes usage scenario for each execution path. The main idea of the algorithm to construct GROUM is to

explore all the execution path in CFG and build temporary GROUMs when exploring those paths. Once a path has been explored, it collects the GROUM that have been built for that path. Each CFG has a start node which is the first instruction and a termination node which is the return node. Our algorithm needs to find all execution paths from the start node to the termination node. One problem that the algorithm needs to consider is to handle loops that occur in the CFG. These loops represent while or for loops in source code. The instructions inside loops may be executed either several times or zero time, thus, could lead to infinity number of execution paths. On the other hand, considering these instructions once can help build usage scenario associated with loops. Therefore, our algorithm consider instructions inside a loop to be executed either once or none. In other words, a loop is executed at most once.

Detailed algorithm. Figure 3.8 show the pseudo-code for our algorithm. Input of the algorithm is the bytecode of a method M. The algorithm starts with creating CFG from the bytecode instructions using techniques described the previous section (line 1). Similar to depth first search algorithm, we maintain a stack F to store states which are frontiers to explore (line 4). Each state of our algorithm represents an incomplete execution path and contains following information: 1) start node: the current node in CFG when a state is pop from stack F, 2) explored nodes: a set of all nodes in CFG that have been visited 3) a temporary GROUM. The initial state S_t is created in line 3 with start node is the begin node of CFG, explored nodes and the temporary GROUM of this state are empty. The stack F is initialized with S_t in line 4. In the main while loop of the algorithm, each time, a state S is pop from F. We start with SN, which is the start node of S (line 8) and explore the path from SN to the next control node in CFG. Whenever SN is a normal node in CFG, we use SN to build and update the temporary GROUM of S (we will describe the algorithm to build a temporary GROUM in the next section). We then add SN to the set of explored nodes of S and update SN equal to the next node of SN in CFG (because SN is still a normal node in CFG, it only has one next node). After the loop from line 9 to line 12. SN

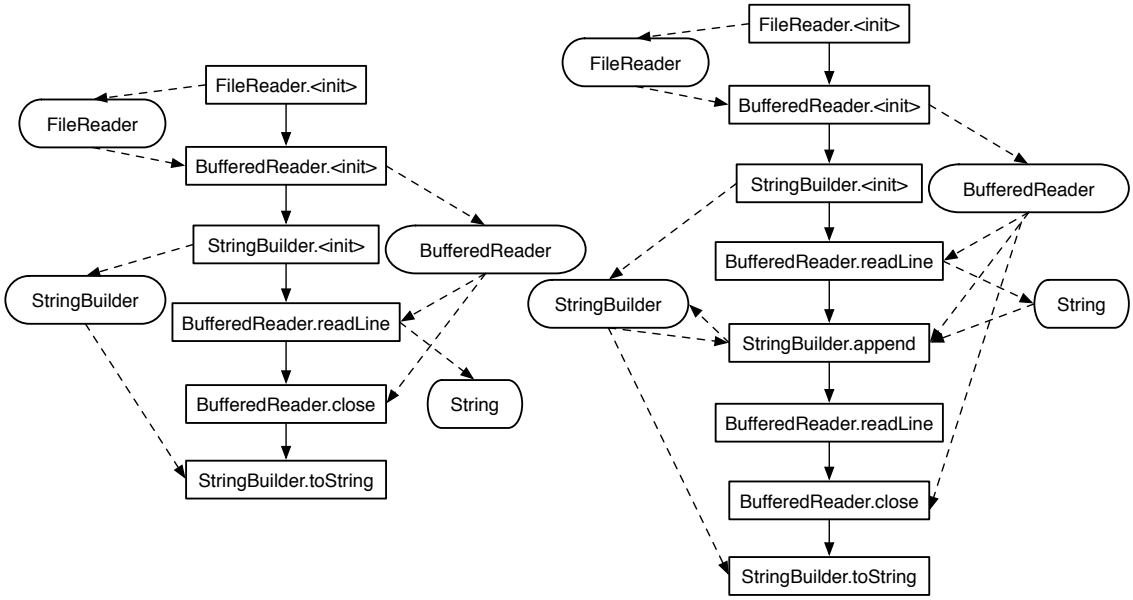

Figure 3.9: GROUMs

now is a control node in CFG. If SN is the return node of CFG then we have explored one execution path of CFG, the temporary GROUM of S now becomes a final GROUM, thus, it is added to the list A (line 14). If SN is not the return node, we need to consider all unexplored out going edges of SN. For each next node NN of SN, if NN is not in the set of all explored nodes of S, we copy all information of S into a new state S_c, set NN is the start node of S_c, and push S_c into F. The algorithm terminates when the stack F becomes empty. Our algorithm explores all the paths of CFG and only go through each loop at most one time. The list A now contains all GROUM of the method M. We next describe algorithm to build temporary GROUM.

Building temporary GROUM. Initially, before considering any instructions, we create an empty GROUM. For each parameter of the method, we create a corresponding object node and add to the GROUM. When an instruction is considered (line 14), it creates a new action node and add that node to the current GROUM. After adding the new action node, we need to update edges. We add a control edge from the last inserted action node to the new action node. For data edges, we first need to add data edges from object nodes that

are parameters of the instruction to the new action node. To do that, we maintain a map $M_{current}$ to store the current mapping between registers and object nodes. When an action node is created, we use $M_{current}$ to identify all object nodes that are used as parameters of the action, and we add data edges from those object nodes to the action node. Consider the instruction 000f: invoke-virtual v0, Ljava/io/BufferedReader;.readLine:() Ljava/lang/String;, it has an input parameter, which is the register v0, using $M_{current}$ we know that when we create an action node from the instruction, v0 is currenty holding a reference to a BufferedReader object node. We then add a data edge from the object node to the newly created action node. An instruction may return a value to a register. If it does not create a new object i.e. it returns a reference of an object has already been created before, then we identify the object node represent that object by $M_{current}$ and add a data edges from the action node to this node. Otherwise, we create a new object, add a data edge from the action node to the new object node and update mappings in $M_{current}$.

3.4.2 API sequence extractor

In this section, we describe how to extract API method sequences for one object or multiple objects from of a method using GROUM. For single object, sequence extractor scans through a GROUM to find object nodes associates with Android API objects. For each object node O_i we find all action nodes that have data edges connected with O_i and sort these action nodes by execution order. From the sorted action sequence, we only consider action nodes that represent API methods to get an API method sequence. For example, in Figure 3.9, for the first GROUM, there are three action nodes related to the object node BufferedReader and all of them are Android API methods, thus, the API method sequence associates the object node BufferedReader is (BufferedReader.init, BufferedReader.readLine, BufferedReader.close).

To extract action sequences that involve multiple API objects, we first identity usage-dependent object sets. For each action node that represents API methods, we collect all

API object nodes that have data-dependent with it, then we form the set of object types correspond to those object nodes. In the first GROUM of Figure 3.9, there are two sets of usage-dependent objects: (FileReader, BufferedReader) has data-dependency with the action node BufferedReader.init, and (BuffedReader, String) has data-dependency with the action node BufferedReader.readLine.After collecting usage-dependent object sets, for each object set, we extract corresponding API method sequences using same technique for single object. For example, the API method sequence for the set (FileReader, BufferedReader) in the first GROUM is (FileReader.init, BufferedReader.init, BufferedReader.readLine, BufferedReader.close).

There should be only one method sequence for an object or a set of usage-dependent objects in each GROUM of a method. Thus, after extracting method sequences in all GROUMs of the method, we remove duplicate sequences and only keep distinctive sequences.

3.4.3 HAPI learner

The HAPI learner uses a collection of API method sequences of an object (a set of usage-dependent objects) to learn the API usages model. In this section, we describe the training algorithm to estimate parameters of HAPI model from training data. We also present a method to choosing the number of hidden states.

Training algorithm

The training algorithm aims to estimate HAPI's parameters $\lambda = (A, B, \pi)$ given a collection of API method sequences. In general, Baum-Welch algorithm is often used to estimate parameters of Hidden Markov Model [90]. In this paper, we present a modified version of Baum-Welch algorithm for the problem of learning API usages. The input of the algorithm is a collection of API method sequences. We observed that there are many method sequences are duplicated in the collection. Thus, to save space and speed up the training algorithm, we store training data as a map, where each method sequence is mapped to the number of times it occurs in the collection. Initially, the parameters of a HAPI are assigned

```
1  function TrainHAPI(TrainSet S, NHiddenStates K)
2    initialize λ = (A, B, π) by random values
3    repeat
4      for (Y_n = (y_1, y_2, ..., y_T), c_n) ∈ S
5        α = Forward(λ, Y_n, T)
6        β = Backward(λ, Y_n, 1)
7        for i ∈ 1..K: {compute state transition probabilities}
8          γ_i^n(t) = α_{i,t} β_{i,t} / Σ_{j=1}^K α_{j,t} β_{j,t}
9          for j ∈ 1..K, t ∈ 1..T − 1:
10           ξ_{ij}^n(t) = α_{i,t} a_{ij} β_{j,t+1} b_j(y_{t+1}) / Σ_{k=1}^K α_k(t) β_{k,t}
11     for i ∈ 1..K: {update model parameters}
12       π_i^{(s+1)} = (1/D) Σ_{n=1}^N c_n × γ_i^n(1)
13       for j ∈ 1..K:
14         a_{ij}^{(s+1)} = (Σ_{n=1}^N c_n × Σ_{t=1}^{T−1} ξ_{ij}^n(t)) / (Σ_{n=1}^N c_n × Σ_{t=1}^{T−1} γ_i^n(t))
15       for v_m ∈ V:
16         b_i^{(s+1)}(v_m) = (Σ_{n=1}^N c_n × Σ_{t=1}^T 1_{y_t = v_m} ξ_i^n(t)) / (Σ_{n=1}^N c_n × Σ_{t=1}^T γ_i^n(t))
17   until convergence
18   return λ
19
20 function Forward(HAPI λ, APISequence Y, Position P)
21   α = {α_{i,t}}   1 ≤ i ≤ K, 1 ≤ t ≤ P
22   for i ∈ 1..K: α_{i,1} = π_i b_i(y_1)
23   for i ∈ 1..K, t ∈ 2..P:
24       α_{i,t} = b_i(y_t) Σ_{j=1}^K α_{i,t−1} a_{ij}
25   return α
26
27 function Backward(HAPI λ, APISequence Y, Position P)
28   β = {β_{i,t}}   1 ≤ i ≤ K, P ≤ t ≤ T
29   for i ∈ 1..K: β_{i,T} = 1
30   for i ∈ 1..K, t ∈ T − 1..P:
31       β_{i,t} = Σ_{j=1}^K β_{j,t+1} a_{ij} b_j(y_{t+1})
32   return β
```

Figure 3.10: Training algorithm

with random values. The main idea of the algorithm is to iteratively estimate parameters to maximize the likelihood function (the probability of generating data given model). The iterative process terminates when the estimated values of parameters converge.

Figure 3.10 shows the algorithm for training HAPI. Its input includes training data S and the number of hidden states K. The parameters of the model are initialized randomly (line 1). Let us describe steps in the main loop of the algorithm:

Step 1. Compute forward and backward probabilities. For each method sequence in training data we use dynamic programming to compute forward probabilities $\alpha_{i,t}$ (using Forward function) and backward probabilities $\beta_{i,t}$ (using Backward function) [90]. $\alpha_{i,t}$ is defined as the probability of seeing partial method sequence $y_1, y_2, ..., y_t$ and being in state q_i at time t

given the model λ:

$$\alpha_{i,t} = P(y_1, y_2, ..., y_t, i_t = q_t|\lambda) \tag{3.1}$$

$\beta_i(t)$ is the probability of seeing the ending partial sequence $y_{t+1}, ..., y_T$ given state q_i at time t and the model λ:

$$\beta_{i,t} = P(y_{t+1}, ..., y_T|i_t = q_i, \lambda) \tag{3.2}$$

Step 2. Compute state probabilities γ and state transition probabilities ξ using forward and backward probabilities. $\gamma_i^n(t)$ is probability of being at state q_i at time t given method sequence Y_n and the model λ:

$$\gamma_i^n(t) = P(i_t = q_t|Y_n, \lambda) = \frac{\alpha_{t,i}\beta_{i,t}}{P(Y|\lambda)} \tag{3.3}$$

The state transition probability $\xi_{ij}^n(t)$ is the probability of being in state q_i at time t and making transition to state q_j at time $t+1$ given method sequence Y and the model λ:

$$\xi_{ij}^n(t) = P(i_t = q_i, i_{t+1} = q_j|Y_n, \lambda) \tag{3.4}$$

Step 3. Reestimate model parameters. In this step, the model parameters are estimated as expected values of probabilities that we computed in the previous step. $\gamma_i^n(t)$ is computed as the expected number of time state q_i is at time 1. $a_{ij}^{(s+1)}$ is estimated as the expected number of transitions from state q_i to state q_j compared to the expected total number of transitions from state q_i. $b_i^{(s+1)}(v_m)$ is the expected number of times the output method have been equal to v_m while in state i over the expected total number of times in state q_i. In the update equations, D is the total number of method sequences in the training set, and $1_{y_t=v_m}$ is the indicator function.

Choosing the number of hidden states

When training the HAPI model for an object (or a set of objects), we need to specify number of the hidden states as an input for the training algorithm, but we often do not know the how many states that the object has. Our idea in choosing the number of hidden

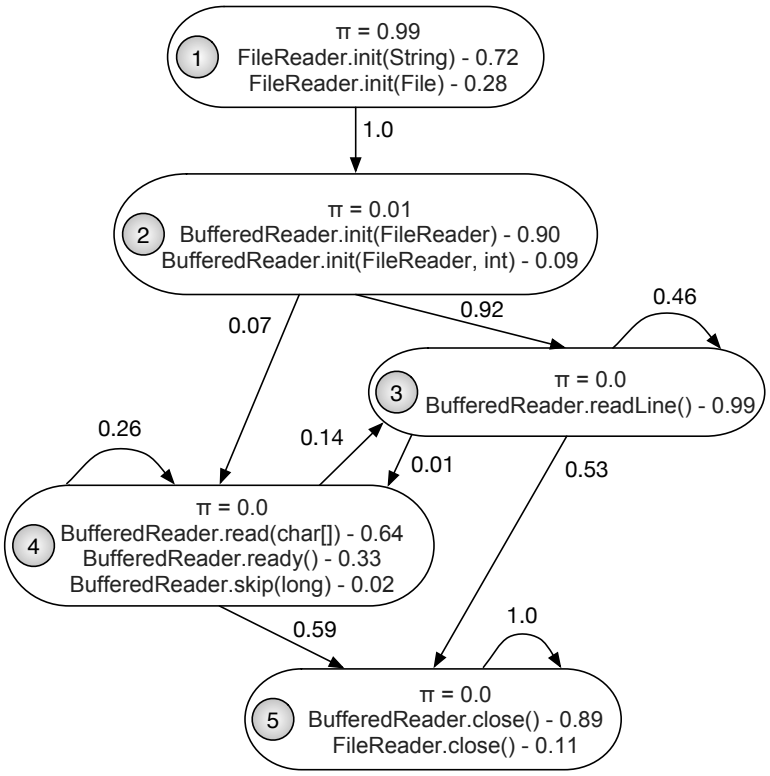

Figure 3.11: HAPI for FileReader and BufferedReader objects

states K is try to build models with different K and find a model that best describe new method sequences. This problem is equivalent to finding K that maximizes the probability of generating new data, i.e. likelihood function. In this method, we divide a training set into two sets, one is used to train models and one is used as validation data to optimize the number of hidden states. For each K in a range, we train a HMM model with K as the number of hidden states. Then, for each model, we compute the likelihood of validation data which is the probability of generating the validation data given the model. We then choose K of the model that maximize the likelihood of the validation set. Figure 3.11 shows graph representation the HAPI model for the pair of objects FileReader and BufferedReader as a result of HAPI learner component. To make it easier to display, we round probabilities and ignore probabilities that is less than 0.01.

```
1  function NextAPICall(HAPI λ, APISequence Y, Location T)
2    R = ∅ // a ranked list of candidates
3    α = Forward(λ, Y, T − 1)
4    β = Backward(λ, Y, T + 1)
5    for v ∈ V:
6      for i ∈ 1..K:
7        α_{i,T} = b_i(v) Σ_{j=1}^{K} α_{j,T−1} a_{ij}
8        β_{i,T} = Σ_{j=1}^{K} β_{j,T+1} a_{ij} b_j(v)
9        score = Σ_{i=1}^{K} α_{i,T} β_{i,T}
10       UpdateCandidateList(R, v, score)
11   return R
```

Figure 3.12: Algorithm for suggesting next method call

3.4.4 API usage recommendation

In this section, we present an algorithm for recommending API method call using HAPI. The input of the algorithm is the HAPI model H of an object (set of objects) and the incomplete API method calls $Y = (y_1, ..., y_N)$ associated with the object (set of objects) in current editing code. The location T of Y is missing. The output of the suggestion algorithm is a ranked list of API methods that could be filled as the method call at position T. The idea of our algorithm is to place each API method as the method call of Y at location T and compute score of this assignment as the probability of generating the updated sequence (including the new API method) given the HAPI model. Then we add the API method with score to the ranked list of all candidates.

Figure 3.12 shows the algorithm. In the first part of our algorithm, we compute forward probabilities at position $T - 1$ (using Forward function). We also compute backward probabilities at position $T + 1$ (using Backward function). Then, we place each API method v as the method call of Y at position T and compute forward and backward probabilities at that position (line 6-8). The *score* of v is the probability of generating sequence $(y_1, ..., y_T = v, ..., y_N)$ is computed by summing all product of forward and backward probabilities:

$$P(Y, y_T = v|\lambda) = \sum_{i=1}^{K} P(Y, y_T = v, i_T = q_i|\lambda) = \sum_{i=1}^{K} \alpha_{i,T}\beta_{i,T}$$

The algorithm returns a ranked list of all the API method candidates with scores for suggestion.

3.5 Code recommendation tool

We develop DroidAssist, a recommending tool for API usages. When a developer is writing code, DroidAssist can analyze the code being written and recommend (and fill-in on request) the next or missing API method calls. To help the developer makes more effective choices, those calls are ranked based on their likelihoods of appearance in the existing code context. DroidAssist can also detect suspicious API usages in existing code (i.e. ones rarely/unlikely to be used) and repair them with more probable usages. More details will be discussed in the next sections. The tool and video demonstration are available at our website [1].

DroidAssist is released as a plugin of Android Studio [34], the standard IDE for Android apps development. After installation, it is incorporated with Android Studio and users can invoke it directly from the current editing view (for method call recommendation) or via the menu (for method sequence validation). This section presents its two usage examples.

3.5.1 Recommending next method call

Assume that the developer wants to write code for a database transaction. She has created a database query that return a Cursor object and made a call to begin the transaction. However, she forgets how to use the returned Cursor object properly. She invokes the built-in code completion engine, but it just lists all methods that can be called on the Cursor, thus does not help her to make an appropriate selection.

Now, with DroidAssist, the developer will have better recommendations. Figure 3.13 shows a screenshot of Android Studio with DroidAssist invoked (via the keystroke Ctrl + Shift + Space) for the current editing code. As seen, DroidAssist displays a ranked list of methods that can be called for the Cursor object. Each method has a score represents the

[1] http://useal.cs.usu.edu/droidassist/

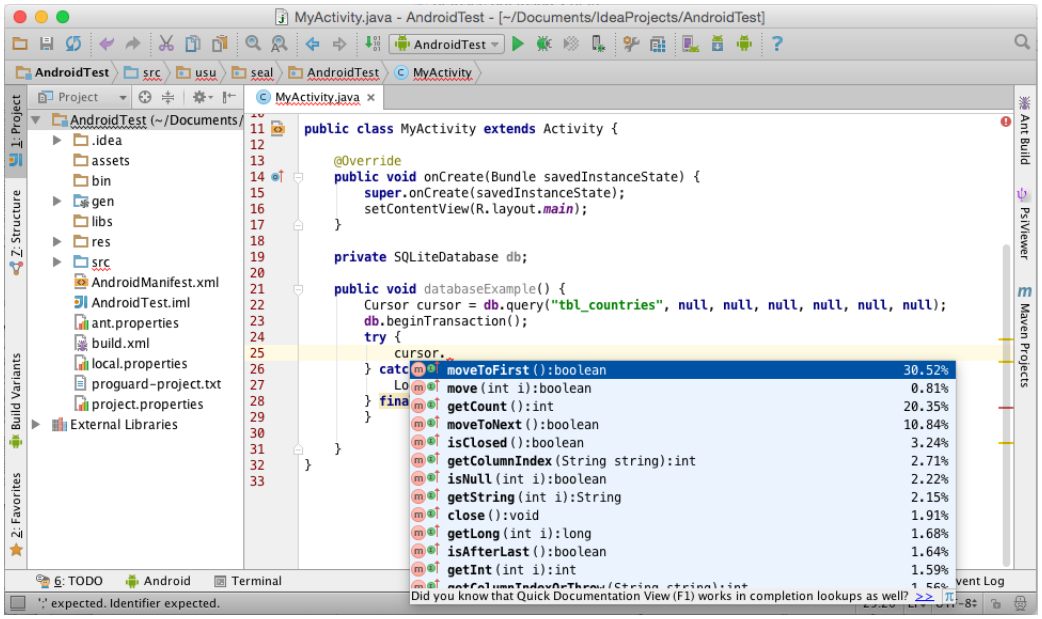

Figure 3.13: Method call recommendation by DroidAssist

probability of how likely it might be called in the given current context of the object and other interacting objects. In this example, method moveToFirst has the highest score of 30.52%. If the developer chooses it, it will be filled in the current position in the editor.

DroidAssist uses code context including the current method calls to infer and recommend the next call. For example, in Figure 3.14, object db already has three method calls: beginTransaction, insert, and setTransactionSuccessful. Thus, DroidAssist predicts that the next method call will be endTransaction with a probability of 61.38%, which is the highest among all available methods for this object.

DroidAssist recommendations are more accurate with more context information. For example, if the method sequence for object db has only two calls: beginTransaction and insert, setTransactionSuccessful has a probability of 15.11%. However, if endTransaction is already added, probability of setTransactionSuccessful increases dramatically to 65.65% (see Figure 3.15).

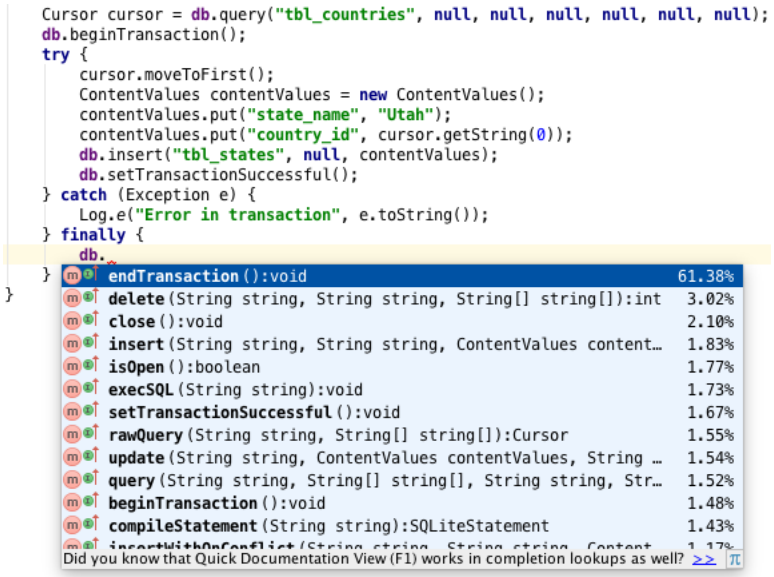

Figure 3.14: Call recommendation for object db

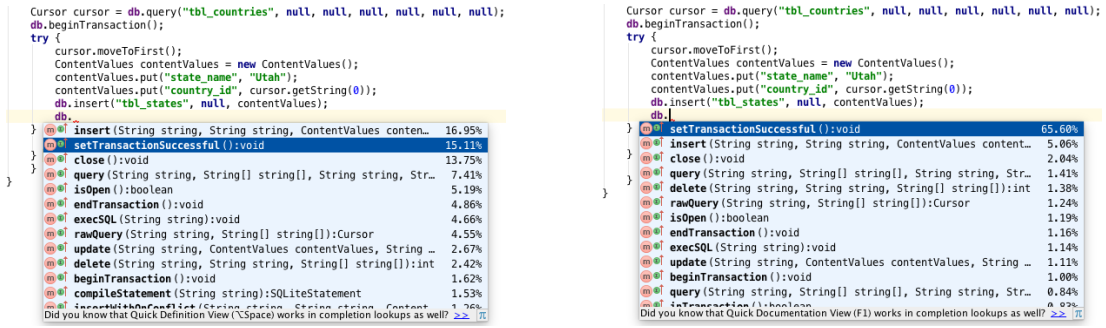

Figure 3.15: API call recommendation before and after calling endTransaction method

3.5.2 Analyzing method sequence

DroidAssist can analyze a given method sequence in existing code report. If it is a suspicious API usages (i.e. is rarely used or unlikely to be used), DroidAssist can offer fixes with more probable method sequences. Figure 3.16 shows a code example involving a MediaRecorder object. After writing code, the developer wants to check if his usage is acceptable. To invoke DroidAssist for that task, she moves to a line of code containing mediaRecorder variable, opens Analyze menu and selects Analyze API Call Sequence.

DroidAssist then will analyze the API usage for that object. Figure 3.17 shows the analyzed result. The left of the dialog shows the original method sequence. The right of the

```
public void mediaRecorderExample(String fileName) {
  mediaRecorder.setAudioChannels(2);
  mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
  mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
  mediaRecorder.setOutputFile(fileName);
  try {
    mediaRecorder.prepare();
  } catch (IOException e) {
    e.printStackTrace();
  }
  ...
}
```

Figure 3.16: Source code examples of using MediaRecorder objects

dialog shows the suggestions for repair. If DroidAssist detects a suspicious method sequence, it will suggest three actions: replace, add, or delete a method call, to make the usage more probable. In the example, DroidAssist detects that calling setAudioChannels at the beginning might not be a proper usage. It recommends to replace that method by setAudioSource. If the user choose option Replace and press Apply, DroidAssist will repair the API call sequence in the code editor by replacing setAudioChannels with setAudioSource.

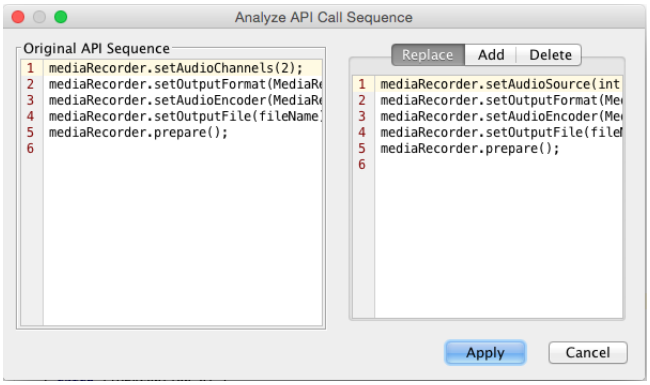

Figure 3.17: Repair suggestion for suspicious usage

3.6 Design and implementation

DroidAssist has five major components including two modules to extract API usages (represented as graph-based object usage models - GROUM [77]) from source code and bytecode, a module to extract method call sequences from those usage models, a module to train HAPIs from those sequences, and two modules that use the trained HAPIs to

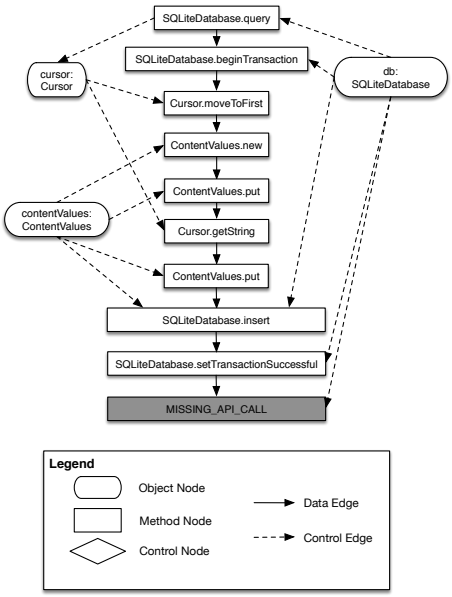

Figure 3.18: A GROUM example

recommend next method calls or check/repair an existing method sequence. Let us describe those modules in more details.

GROUM builder

DroidAssist uses GROUM (Graph-based Object Usage Model) [77], developed by the fourth author and his colleagues to represent the raw API usages in source code and bytecode. Thus, to collect training data, SALAD has a module to extract GROUMs from bytecode of existing Android apps and a similar module to extract GROUMs from the code being written, which are used for its two tasks of method call recommendation and validation. Figure 3.18 shows an example of GROUM built for the code snippet in Figure 3.14.

API method sequence extractor

Because HAPI is designed based on Hidden Markov Model, it works on sequences, not graphs. Thus, DroidAssist has a module named API Sequence Extractor to extract sequences of method calls from the GROUMs. For a given GROUM, this module simply traverses all its paths and extracts the method calls along those paths. However, to ensure the extracted

method sequences are meaningful, it only keeps the sequences involving the same API objects (i.e. have data dependency). More details can be found at [83].

HAPI learner

This module is responsible for training HAPI models from the extracted method sequences. It first sorts those method sequences by the involving API objects. For example, all method sequences involving the usages of a single MediaRecorder object are grouped together and then will be used to train the HAPI representing the usage of a single MediaRecorder object. Sequences involving the usages of a Scanner and a StringBuilder object are grouped and used to train the HAPI for those two objects.

DroidAssist uses a modified version of Baum-Welch algorithm to train a HAPI [83]. In the training process, it figures out the optimal number of internal state of the HAPI and estimates all probabilities involving the HAPI's states (i.e. how likely a state is selected as the starting state, how likely the object changes from a state to another) and method calls (i.e. how likely a method is called when the object is in a state). Please refer to [83] for more details of this algorithm.

Method call recommender

This module provides the recommendations of the next method call for the currently editing code (and performs the completion task if requested). Its input is a collection of method sequences provided by API Sequence Extractor along with the position of the missing call. It places every available method m at the missing position and computes the probability of the newly created sequence based on the HAPI of the API objects involving that sequence. It then combines those probabilities into a final score for method m. All methods then are ranked by those scores and proposed to the developer, as seen in Figure 3.14.

Table 3.1: Data collection

Number of apps	207,603
Number of classes	32,080,884
Number of methods	59,636,164
Number of bytecode instructions	1,759,540,508
Space for storing .dex files	689.8 GB

Method sequence analyzer

This module evaluates an API call sequence. Its input is a method sequence for one or more API objects. This module uses the corresponding HAPI to compute the probability of this sequence. If that probability is smaller than a threshold, the sequence is considered to be suspicious. If that happens, the Method Sequence Analyzer will explore different modifications on the sequence (i.e. by adding/replacing/removing a method call) to find more probable sequences and offers them for repair.

The threshold is computed based on the distribution of probabilities of all unique method sequences used for training the HAPI. Based on our experiment, we currently use the 75% percentile as the threshold. That means, if the HAPI is trained with 100 unique sequences, and 75 of them have probabilities smaller than 0.1, this value is chosen as the threshold.

3.7 Empirical evaluation

We conducted several experiments to study the run-time performance of Salad and compare the accuracy of HAPI models and baseline models in recommending API method calls. All experiments are executed on a computer running 64-bit Ubuntu 14.04 with Intel Core i7 3.4Ghz CPU, 16GB RAM, and 1TB HDD storage.

3.7.1 Data collection

Table 3.1 summarizes data collected for our experiments. In total, we downloaded and analyzed **207,603 apps** from 26 categories in Google Play Store. We only downloaded apps with the average rating of at least 3 (out of 5). This is based on the assumption that the

Table 3.2: Extracting API method sequences

	Single object	Multiple object
Number of distinct object types (sets)	4,877	43,408
Total number of method sequences	195,692,154	143,678,399
Average number of method sequences	40,125	3,309
Average length of method sequences	3.32	7.72

high-rating apps would have high quality code, and thus, would contain API usages of high interest for learning.

Since Android mobile apps are distributed as .apk files, SALAD unpacked each .apk file and kept only its .dex file. The total storage space for the .dex files of all downloaded apps is around 700 GB. SALAD parsed those .dex files and obtained **32 millions classes**. It analyzed each class and looked for all methods in the class to build GROUM models. Since an Android mobile app is self-contained, its .dex file contains bytecode of all external libraries it uses. That leads to the duplication of bytecode of shared libraries. Thus, SALAD maintains a dictionary of the analyzed methods, thus, is able to analyze each method only once. In the end, it analyzed **60 millions methods** which have in total nearly **1.8 billions** bytecode instructions.

3.7.2 Extracting API method sequences

SALAD built GROUM for each remaining method in the dataset and extracted API method sequences from those models. It extracted sequences for both single object usages and multiple object usages. Since we focus on learning Android API usages, only sequences involving classes and methods of Android application framework were extracted. Sequences with only one method call were disregarded.

Table 3.2 summarizes the extraction result. In total, SALAD has extracted nearly **195 millions method sequences** involving single object usages of more than **4,800** classes. There are more than **43,000** distinct usage-dependent object sets made from those types and SALAD has extracted over **143 millions method sequences** involving them. The running

Table 3.3: Training HAPI models

	Single object	Multiple object
Number of trained models	3,042	21,526
Average number of hidden states	14.07	15.65
Total training time	1h 20m	2h 30m
Total space to store trained models	30.3 MB	134.1 MB

time and space is efficient. SALAD took totally 28 hours to build GROUMs and extract method sequences for 200 thousands apps, i.e. **0.5 seconds per app** on average. It needed less than 1.5 GB of working memory.

3.7.3 Training API usage models

Once all API method sequences are extracted and stored, SALAD trained HAPI models for each object type or usage-dependent object sets. For example, a HAPI is trained for the usages involving any MediaRecorder object and another HAPI is trained for the usages involving any two objects {FileReader, BufferedReader}. However, some HAPIs have too few sequences for training, making the training procedure unstable. Therefore, we do not train usage models for API objects with less than 10 method sequences. Overall, we discarded less than 237,170 sequences out of 350 millions.

The training process for the remaining models is summarized in Table 3.3. As seen in the table, SALAD is time- and space-efficient. It trained nearly 24,000 HAPI models in about 3 hours 50 minutes i.e. **0.55 second/model** on average. The total storage for all of them is of 160 MB, i.e. **7 KB/model** on average.

Table 3.2 and Table 3.3 show that SALAD can train HAPI models for 60% (3,042/4,877) of encountering API single object usages and 50% (21,526/43,408) of multiple object usages. However, untrained APIs are rarely used, each has less than 10 usages in 207,603 apps (59,636,164 methods).

3.7.4 API usage recommendation

We performed an experiment to measure the accuracy of HAPI in recommending API usages and compare to baseline models. In this experiment, we chose the task of recommending the next method call. That is, given an API method sequence and the model is expected to recommend the most probable next method call. This task has been used in the evaluation of prior approaches [70, 11].

In the experiment, we predicted and evaluated *all method calls* in every method sequence from the testing set. For a method call c_i at position i, we provided its $i-1$ prior method calls as the input and used the model to infer the top-k most probable next method calls $R_k = \{r_1, r_2, ..., r_k\}$. If c_i is in R_k, we consider it as a hit, i.e. an accurate top-k recommendation. The top-k accuracy is the ratio of the total hits over the total number of evaluated method calls.

Baseline models

We chose n-gram and recurrent neural network (RNN) trained for method call sequences as two baseline models for comparison due to the following reasons. First, both of them are statistical models, thus, is comparable to HAPI, which is also a statistical model. In addition, n-gram is widely used in recent research on code completion [39, 76]. More importantly, Raychev *et al.* recently evaluated RNN and n-gram and reported RNN as the best approach.

n-gram model is a simple statistical model for modeling sequences. An n-gram model learns all possible conditional probabilities $P(m_i|m_{i-n+1}...m_{i-1})$, where m_i is the current method call and $m_{i-n+1}...m_{i-1}$ is the sub-sequence of $n-1$ prior method calls. This is the probability that m_i occurs as the next method call of $m_{i-n+1}...m_{i-1}$. Using the chaining rule, we can use an n-gram model to compute the generating probability of any given method sequence $m_1...m_n$. In our experiment, we used a 3-gram model (i.e. the occurrence probability of a method call depends on its two previous calls) as used in prior work [92]. We also implemented Witten-Bell smoothing [112] technique for this model.

Recurrent neural network (RNN) is a class of neural networks for learning sequences. Like a HAPI, a RNN can be trained with a collection of method sequences and then is able to compute the probability of the next method call for any given method sequence. In other words, RNN can compute all conditional probabilities $P(m_i|m_1...m_{i-1})$ for any given a method sequence $m_1...m_n$. To do that, it maintains a context vector c_i represents current context of sub-sequence up to $m_1...m_{i-1}$. A function f is learned from data to compute the context vector at position i, $c_i = f(m_i, c_{i-1})$ given the current method call m_i and previous context c_{i-1} while another function g is learned to compute the probability of the next call m_{i+1}, $P(m_{i+1}|m_1...m_i) = g(c_i)$ given the current context c_i. In our experiment, we used a publicly available implementation of RNN[2] with the number of hidden states of 40 which is also used in prior work [92].

Experiment results

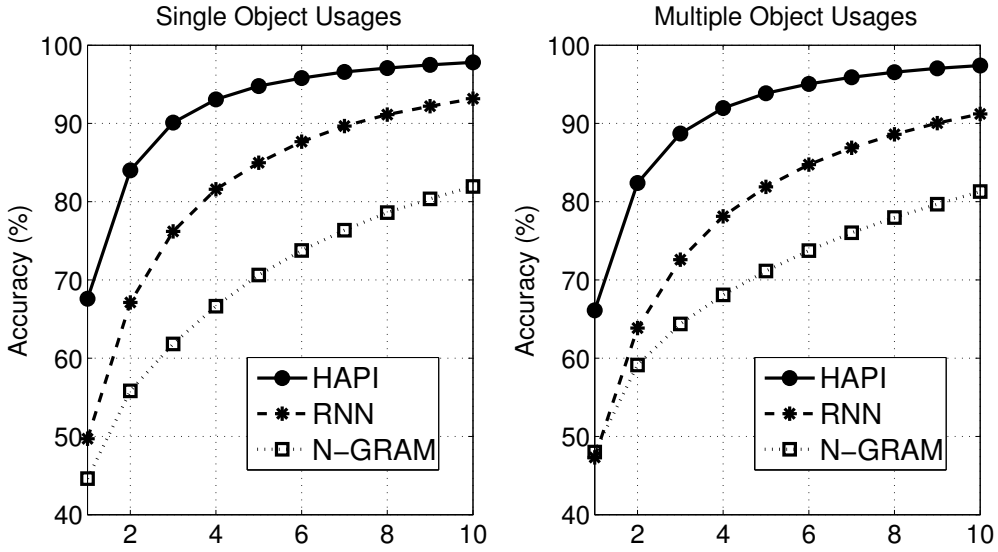

Figure 3.19: Accuracy of next method call recommendation

Our experiment is a 5-fold cross validation. That is, for each object type or object set in the experiment data, we divided its method sequences into five equal folds. HAPI and

[2]http://www.rnnlm.org

two baseline models are trained in four folds and tested in the remaining folds (i.e. they are trained and tested in the same data). We repeated this process five times, each for an individual fold as test data and computed the average accuracy of each model in five iterations as its final result. We chose 5-fold cross validation over the popular 10-fold cross validation to reduce experiment time because RNN is very time-consuming.

Figure 3.19 shows the experiment results for method sequences extracted from bytecode of 200 thousand apps downloaded from Google Play. As seen in the charts, all three models have consistent accuracy for single and multiple object usages. More importantly, HAPI can recommend API usages with very high levels of accuracy. For example, for single object usage, it has a top-3 accuracy of 90% and a top-10 accuracy of 98%. In addition, HAPI significantly outperforms RNN and n-gram models. For example, the corresponding top-3 and top-10 accuracy of n-gram model are 62% (28% lower) and 82% (16% lower). The top-3 and top-10 accuracy of RNN model are 76% (14% lower) and 93% (5% lower). On average, HAPI has 11% improvement over RNN and 21.5% improvement over n-gram.

3.8 Discussion

Table 3.4: Experiment settings in recent research about statistical language models for code

Approach	Model	Code token	Code form	Training	Testing
SALAD	3-gram, RNN, HAPI	Method call (Android)	Bytecode	59,636,164 methods	5-fold cross validation
SLANG [92]	3-gram, RNN	Method call (Android)	Source code	3,090,194 methods	20 pre-selected examples
GraLan [71]	9-gram	Method call (Java)	Source code	1,000 projects	5 independent projects
SLAMC [76]	3-gram	Semantic token	Source code	9 (Java) + 9 (C#) projects	10-fold cross validation
Tu et. al [104]	3-gram (cache)	Lexical token	Source code	9 (Java) + 9 (Python) projects	10-fold cross validation
Hindle et. al [39]	3-gram	Lexical token	Source code	5 projects	200 held-out source files

Occam's razor principle could explain HAPI's improvement over n-gram and RNN. To model usages involving M methods, a HAPI with K hidden states uses $K + K^2 + KM$ parameters, while an n-gram uses M^n parameters (i.e. all possible sub-sequences of n method calls), and a RNN of H hidden nodes uses $MH + H^2 + HM$ parameters. Since K is often much smaller than H and M, HAPI has less parameters thus easier to train and generalize.

Chapter 4

Code Recommendation Model for Exception Handling

4.1 Introduction

Exceptions are unexpected errors occurring while a program is running. If not handled properly, exceptions could lead to serious problems such as system crashes or resource leaks. For example, when a program tries to parse an integer from a string but the string does not represent a valid number, a NumberFormatException exception occurs. If the program ignores that exception, it will crash. Thus, effective exception handling is important in software development. A prior study [108] reports that correctly handling exceptions could improve 17% in the performance of software systems.

Most modern programming languages provide built-in support for exception handling. For example, in Java, we can wrap a try block around a code fragment where an exception might occur and add a catch block to handle that exception. However, programmers have to determine themselves what code fragment to protect with the try block, what exception type to handle in the catch block, and what to react when such an exception occurs.

Learning to handle exceptions properly is often challenging due to several reasons. First, modern software development relies strongly on API libraries. An API library often defines many specific exception types and exception handling rules. For example, in Java SDK, class FileNotFoundException is defined for the *'file not found'* exception. When such an exception happens, the software system could react by notifying users about the error and writing the relevant information (timestamp, filename, caller, etc.) to a system log for future debugging.

However, a major API library often consists of a large number of components. For example, the Android application framework contains over 3,400 classes, 35,000 methods, and more than 260 exception types [58]. Thus, it is very difficult to learn and memorize what

method could cause what exception and what to react when a particular exception occurs. Secondly, popular API libraries are often upgraded quickly. For example, in 10 years from 2008 to 2018, 28 major versions of Android SDK have been released. The latest version of Java SDK is JDK 12 is released in 3/2019. Each of those major versions often includes very large changes, which could introduce new exceptions and new rules for exception handling.

Most importantly, the documentation for exception handling is generally insufficient. Kechagia *et al.* [51] found that 69% of methods in Android SDK have undocumented exceptions and 19% of the crashes could have been caused by insufficient documentation. Coelho *et al.* [25] found that documentation for many explicitly thrown runtime exceptions (or often called unchecked exceptions) is rarely provided.

Modern code editors can enforce catching of checked exceptions. For example, in Java, FileNotFoundException is a checked exception. Thus, when a new FileInputStream object is created to read from a file, Eclipse will ask the programmer to catch the FileNotFoundException (or throw it). However, such support is not available for runtime (unchecked) exceptions like IllegalArgumentException. In addition, popular code editors cannot suggest the reactions when a checked or unchecked exception happens.

We introduce FuzzyCatch, a code recommendation plugin for Android Studio with advanced support for exception handling. Based on fuzzy logic rules learned from thousands of high-quality programs publicly available in app stores, FuzzyCatch can predict if a runtime exception potentially occurs in a code snippet. Then, as the programmer requests, it can generate the try-catch statement with catch block containing code to catch that exception and recover from it.

The key idea of FuzzyCatch is based on the observation that exception handling can be expressed by two general programming rules: i) *if call method m then catch an exception e*; and ii) *if an exception E occurs then perform action r to react.* For example, if you call the method Activity.unregisterReceiver then you have to catch an IllegalArgumentException. Or, if you call the Integer.parseInt then you have to catch an NumberFormatException. And,

when an SQLiteDatabase occurs, you call method Cursor.close as a simple reaction to close the resource.

FuzzyCatch learns such rules from a large collection of high-quality code. For example, assume that no available documentation specifies that if we call method m, we need to catch an exception of type E. However, we observe from thousands of high-rated apps in Google Android Appstore that whenever m is called, 90% of the times an exception of type E is also caught. Such a high level of co-occurrence suggests a rule that if we call m, we should catch E. However, because those rules are likely *imperfect*, fuzzy logic is used to represent and combine them.

We have conducted several experiments to evaluate the usefulness and effectiveness of our tool. We collected a dataset containing over 21 million methods from over 13,000 highly rated mobile apps in Google Android Appstore. The 10-fold cross-validation of FuzzyCatch on this dataset are promising. FuzzyCatch has the top-1 accuracy of 77% for recommending exception types. That means, in 77% of the cases, the first exception type that FuzzyCatch recommends catching is the one used by app developers. The top-1 accuracy of recommending reaction code is of 70%.

We collected a second dataset of 1,000 real exception related bugs. In its strictest mode, FuzzyCatch flags 734 cases (73%), i.e., suggesting they need exception handling. In its loosest mode, FuzzyCatch flags 821 cases (82%). However, the loosest mode would make around 35% unwanted warnings on code that programmers have never added exception handling code. The strictest mode would make only 7% of unwanted warnings. On the task of recommending handling actions, FuzzyCatch provides meaningful recommendations in roughly 65% of the bug fixes and outperforms the baselines significantly.

4.2 Empirical study

To understand the nature of such exception-related bugs and how programmers fix them, we performed an empirical study of exception handling bugs and fixes. We focus on two main research areas regarding exceptions handling.

1. Causes and effects of exceptions. Missing exception handling code or failing to handle exceptions properly could introduce serious errors on the system. Thus, we focus on how exception bugs affect the running apps. Secondly, we study exception bugs in several aspects including what are object types and methods often cause exception bugs, what are exception types that often being thrown in those bugs, and are there any correlations between object types or methods that cause exception and exception types. The results provide insights on exception bugs.

2. Handling exceptions. In this research area, we focus what developers do in exception bug fixes. Swallowing exception by adding a simple `try-catch` block could avoid the program crashing when exception occurs but might introduce new bugs to the program. Thus, we study whether this type of handling occurs in the bug fixes. Next, we focus on what type of actions developers do in their handling code such as closing or releasing objects that hold resources, creating new objects, etc.

4.2.1 Methodology

Next, we describe how we collected and built the dataset used in our empirical study. The dataset includes real bug fixes that related to exceptions. In particular, we focus on bugs that are caused by not catching exceptions or adding proper exception handlers. For convenience, we defined those bugs as **exception bugs**. The fix for those type of bugs is called **exception bug fixes**.

We first collected a dataset consists of several open-source Android projects. Table 4.1 lists 10 subject projects used in our extraction process. Each project is an application

Table 4.1: The empirical dataset

Mobile App	Website	Commits	Bug Fixes	Candidate Bug Fixes	Exception Bug Fixes
AntennaPod	antennapod.org	3,404	767	89	28
ConnectBot	connectbot.org	1,450	327	39	8
Conversations	conversations.im	3,318	922	123	23
FBReaderJ	fbreader.org	27,944	1,403	118	30
K-9	github.com/k9mail	7,254	1,797	223	31
MozStumbler	location.services.mozilla.com	2,667	692	55	7
PressureNet	cumulonimbus.ca	1,017	203	31	11
Signal	whispersystems.org	2,754	1,030	130	28
Surespot	surespot.me	1,590	274	54	8
WordPress	apps.wordpress.org	15,546	3,691	425	72
Total		**66,944**	**11,106**	**1,287**	**246**

published in Google Play Store [1]. AntennaPod is an open-source podcast manager for Android. ConnectBot is a SSH, telnet and terminal emulator. Conversations is a XMPP-based instant messaging client. FBReader is an e-book reader. K-9 is email client focused on managing large volumes of email. MozStumbler is a wireless network scanner developed by Mozilla. PressureNet is a crowd-sourced barometer network. Signal and Surespot are secure instant messaging apps. WordPress is the official Android client of WordPress. All the projects are written in Java and have source code repositories available on GitHub. For each project, we checked out its source code repository to retrieve all the code and commits. The number of commits of each project (at the time we checked out the repositories) are listed in column Commits. To ensure the reliability of our dataset, we selected medium to large projects compared to other Android projects. Each project has at least over 1,000 commits and the total number of commits are over 66,000.

The first step to produce the dataset is identifying bug fixes. We define a bug fix is a commit in which the developer fixed a bug found in the project. Identifying all bug fixes in a software project is often a hard task as issue tracking systems are incomplete to capture all bugs, and the developers might not describe a fix explicitly. Generally, bug fixes are identified based on two types of bugs: 1) bugs reported through issue trackers, which are called *reported bugs*, and (2) those not reported to issue trackers, which are called *on-demand bugs* [115].

[1] play.google.com

Reported bugs. All the projects have an issue tracking system to track various issues including bugs, improvements, new features, tasks, etc. Using the system, users can report issues or requests they encountered when they are using the software. Each reported issue has an associated issue number and can be labeled based on the type of it (*e.g.* bugs, improvements, new features, tasks, and sub-tasks). As an example, in Wordpress, an user reported a bug with the title "Crash report 3.7: IllegalStateException in WPDrawerActivity #2372". The issue number for this bug is #2372 and it is labeled as [Type] Bug. In the fix commit for the above bug, the commit's message says "fix #2372 by catching IllegalStateException" to indicate that, in the commit, the developer fixed the bug in the issue #2372 by catching IllegalStateException. As the developer already fixed the bug, he marked the status of the issue as closed or fixed. To identify bug fixes from *reported bugs*, [115, 53] used a pattern to extract all commits that mention a issue number. Then they used the issue number to determine whether it indicates a bug or not (using the label of the issue). If the issue is a bug, the associated commits are considered bug fixes. For instance, the commit in the previous example points to the issue #2372, and as the issue is a bug, the commit is considered as a bug fix.

On-demand bugs. Issue trackers often do not contain all information about the bugs and fixes. In many cases, developers might bypass the issue trackers, especially when they discover bugs from other sources by themselves or other sources (e.g. user reviews, discussions with other developers). In such cases, they often fix the bugs and commit the changes without creating an issue. When they commit a change, programmers may write a message to describe the fix. For example, in the project Wordpress, the message of a commit indicates "Fixes crash where postID could be larger than max int value". In the commit, the developer added a try-catch block to handle a NumberFormatException which caused the app scraching if postID is larger than maximum integer value. The bug that the developer referred in the commit message is an *on-demand bug*. To identify bug fixes from *on-demand bugs*, a number of previous studies (e.g. [115, 89, 65] used a simple keyword-based technique. They

identified a commit is a bug fix if its message contains words such as "fix","bug", or "patch". The method is based on assumption that when fixing a bug, developers often write commit message to describe the fix. In the example, as the commit message contains the word "fixes", it is considered as a bug fix.

We identified bug fixes using both methods described above and combine the result to form a set of bug fixes for each project. The column Bug Fixes lists the number of bug fixes of each project. In total, there are over 11,000 bug fix commits over 10 projects.

The next step is identifying exception bug fixes from all bug fixes. Not all bug fixes are related to exception and manually identifying exception bug fixes from all bug fixes is very time consuming. Thus, we used a semi-automated method for identifying exception bug fixes. First, we developed a simple filter technique to only consider bug fixes that can potentially be exception bug fixes. As most of the exception bug fixes involving adding try-catch blocks, we consider a commit to be *candidate* exception bug fixes if the changes in the commit contain at least one adding of catch statement. For example, a change of an example bug fix is shown in Figure 4.1.

COMMIT MESSAGE: "fix #2695: re-introduce a workaround we were using in previous versions"

```
- postContent = new SpannableStringBuilder(
-                                 mEditorFragment.getSpannedContent());
+ try {
+     postContent = new SpannableStringBuilder(
+                                 mEditorFragment.getSpannedContent());
+ } catch (RuntimeException e) {
+     // A core android bug might cause an out of
+     // bounds exception, if so we'll just use the current editable
+     // See https://code.google.com/p/android/issues/detail?id=5164
+     postContent = new SpannableStringBuilder(
+         StringUtils.notNullStr((String) mEditorFragment.getContent()));
+ }
```

Figure 4.1: An example bug fix

After using this filtering method, the number of *candidate* exception bug fixes are 1,287 as showed in column Candidate Bug Fixes of Table 4.1.

Next, we manually inspected the remaining 1,287 bug fixes to find exception bug fixes.We formed an assessment group that consists of three Ph.D. students in the field of software

engineering to do this task. A bug fix is considered an exception bug fix if an exception is thrown on the bug and the developers fix the bug by adding proper exception handling code. Our criteria to determine whether a bug fix is an exception bug fix is based on commit message, changes in commit, and comments adding in source files. Figure 1 shows an exception bug fix. We recorded several information related to exception bug fixes for the purpose of our study, including the effects of the exception bugs, what types and methods causing exception, the exception types, the handling action of the fixer, the logging messages, etc. Finally, we identified 246 exception bug fixes across 10 projects. Column Fixes of Table 4.1 shows the number of exception bug fixes for each projects.

4.2.2 Empirical results

Finding 1. First, we focused on how exception bugs affect the running apps. Thus, we classified exception bugs into different types of effects including: CRASH, UNSTABLE, UNKNOWN. CRASH implies that the app crashed when an exception bug occurred. UNSTABLE means that the app continued to run but in an unstable state or some features might not function properly. If we did not have enough information to figure out the effect of an exception bug, we labeled it as UNKNOWN. We studied the number of exception bugs classified by the defined types. Exception bugs caused apps crashing in over 80% of the cases (199/246). Over 13% (33/246) of the exception bugs caused the app running in a unstable state or some features might not function properly. There are 14 exception bugs are labeled as UNKNOWN as we did not have enough information to classify those bugs in one the two types above. From the statistics, we can see that exception bugs often cause serious problems for the apps such as crashing or running in a unstable state.

Finding 2. Most of exception bugs are caused by Android API methods. Table 4.2 shows top-10 types by the number of times the methods of those types cause exceptions. Our first observation is that all 10 classes are Android APIs (to save space, we do not show the fully qualified name of each class). Further investigated, we found out that 51% (127/246) of

exception bugs is caused by Android API methods. Note that, in our study, we only identified which methods cause exceptions inside the try block of a bug fix. A third-party method that causes an exception may be used by several Android API objects and methods inside its implementation, and the actual cause of exception might be from those API calls. Thus, in reality, we believe the percentage of exception bugs caused by Android APIs could be even higher. The finding suggests that an exception-handling recommendation model should focus on handling exceptions related to Android APIs.

Finding 3. Most of exceptions in bugs are runtime exceptions. Table 4.3 shows top-10 exception types appear in the exception bug fixes. From the table, we have several observations. First, the java.lang.Exception class appears most often. This result could be explained as the fact that when catching exception, developers may not know or care about what type of exception is thrown and they just catch the most general exception. Another observation is that most of remaining exception types are runtime exceptions. There is only one type of runtime errors in the ranked list, which is OutOfMemoryError. Further investigation on the type of exceptions, we find out that in 246 exception bugs, there is 58.13% (143/246) of runtime exception, 7.31% (18/246) of checked exceptions, and 9.34% (23/246) of runtime errors. For the 62 remaining exception bug fixes, developers catch exceptions using general exception types such as Exception or Throwable, thus, we cannot identify types of exceptions in these fixes. The result indicates that the majority of exceptions in the dataset are runtime exceptions.

Finding 4. In this finding, we study what developers do in exception bug fixes. Based on the fixes of developers in the dataset, we classify 3 types of fixes that the fixers performed: catch the exception and do not perform any actions (SWALLOW), re-throw another exception to transfer handling jobs to other functions (RETHROW), invoke method calls or operations to handle exceptions (HANDLING). After analyzing each type of actions, we found that, in total, programmers did not perform any actions to handle exception in about 16% (40/246)

bug fixes. They re-threw another exception in about 12.6% (31/246) of the cases. Finally, fixers handled exceptions in 41.86%(103/246) of the exception bug fixes

Table 4.2: Top-10 types

Types	
android.app.Activity	14
android.content.Context	8
java.lang.Integer	5
android.database.sqlite.SQLiteDatabase	5
android.graphics.BitmapFactory	5
android.graphics.Bitmap	4
java.text.SimpleDateFormat	4
android.content.ContentResolver	3
android.database.sqlite.SQLiteDatabase	2
android.media.MediaPlayer	2
android.database.Cursor	2

Finding 5. In totals, programmers invoke method calls or operations to handle exceptions in 41.86%(103/246) of bug fixes. We label a bug fix as HANDLING if the catch block in that fix contains at least one statement other than a log statement. To further understanding how programmers handle exceptions, we categorized handling actions into several categories. We found that about 52% (54/103) of these bug fixes, programmers used default values to handle exceptions. One example of this handling type is if an exception occurs inside a method that returns an object, in the catch block of the fix, programmers handle exception by returning null. Or if exceptions occur while getting or creating object, he or she will assign the result variable to a default values such as null, true, 0, etc.

Programmers invoke method calls to handle exceptions in the remaining 47 bug fixes. After analyzing these bug fixes, we found that in almost all of these bug fixes (40/47), programmers invoke method calls of the same class with the method that causes exception.

Table 4.3: Top-10 exceptions

Exceptions	
java.lang.Exception	54
java.lang.NullPointerException	25
java.lang.IllegalArgumentException	22
java.lang.OutOfMemoryError	19
android.content.ActivityNotFoundException	14
java.lang.NumberFormatException	9
java.lang.IllegalStateException	9
java.lang.Throwable	8
android.database.sqlite.SQLiteException	8
java.lang.ClassCastException	8

4.3 Motivation examples

Although, most modern programming languages provide built-in support for exception handling. Programmers still have to determine themselves what code fragment to protect with the try block, what exception type to handle in the catch block, and what to react when such an exception occurs. The previous studies [30, 85] shows that programmers might easily forget to handle an exception that potentially occurs in a code snippet. Such mistakes could cause serious issues to the developing app, i.e. crashing. For example, Figure 4.10 (Section 5) shows a serious bug caused by not catching a RuntimeException.

Furthermore, the studies also suggest that in practice, novice and even experienced programmers might not always handle exceptions properly. For example, Figure 4.2 shows an exception related bug fix in the WordPress mobile app. The programmer calls method Activity.unregisterReceiver() with variable mediaUpdate passed as its argument. At runtime, this call causes an IllegalArgumentException and crashes the mobile app. This error is fixed by a try catch block that handles that exception. However, the catch block has no code. Although this bug fix makes the app not crash again, it is a bad programming practice to catch an exception and do nothing.

Figure 4.3 shows another exception related bug fix of Cursor object. In this example, an exception occurs when using Cursor object. To handle this bug, the programmer added a

```
- unregisterReceiver(mediaUpdate);
+ try {
+     unregisterReceiver(mediaUpdate);
+ } catch (IllegalArgumentException e) { }
```

Figure 4.2: An example of swallowing an exception

try-catch block to cover the code portion that uses the Cursor object, and returned null in the catch block. This action of handling may lead to memory leak bugs. After exception occurs and the function returns, the Cursor object still lives and holds system resources until it is collected by the garbage collector. This may prevent other parts of the program to access the resources. The proper way to handle this exception is calling close method on the cursor object before the return statement. The close method will release resources that hold by the cursor object before the function lost the reference to it.

COMMIT MESSAGE: "catch exception when reading message id from database."

```
. . .
+ try {
     if (cursor.getCount() == 0) {
         return null;
     } else {
         cursor.moveToFirst();
         return new Pair<>(cursor.getLong(), cursor.getString(1));
     }
+ } catch (Exception e) {
+     return null;
+ }
```

Figure 4.3: An exception related bug fix

The examples indicate the need exception handling recommendation tools for programmers. Such programming tools could warn programmers about potential exceptions that might occur in the code and assist programmers to handle the exceptions correctly.

4.4 Approach

In this section, we discuss the key techniques of FuzzyCatch. We will start by introducing XRank, a fuzzy logic system to rank and suggest exception types to catch given

a piece of code. Then we describe XHand, a fuzzy logic system for recommending repairing actions when a particular exception occurs in that piece of code.

4.4.1 Recommending exception types

We formulate the problem of recommending exception types as the following: Given a code snippet C, determine of the runtime exception(s) e with a high possibility to occur when C is executed.

Our key idea to solve this problem is to learn from the exception handling code already written in high-quality software systems. We first collect a large collection of high-quality code D. For each method m called in C, we look at all every call of m in D. If those calls often associate with catching an exception e then it is likely that calling m could cause e. Since there might be several methods called in C and several exceptions associating with them in D, we need to combine and rank those exceptions.

This is an example from our experiment data. As discussed in detail in Section 5, we collected roughly 13,000 highly-rated Android mobile apps. In those apps, we found 38,218 calls to method Integer.parseInt. Among them, 12,469 calls (32.6%) appear in try-catch blocks catching NumberFormatException. And this is the exception most co-occurred with Integer.parseInt. Thus, even without documentation, we can infer that calling Integer.parseInt could cause NumberFormatException. Of course, Java API documentation comfirms that.

Similarly, we found that, among 23,406 method calls to Cursor.moveToFirst, 8,159 calls (34.8%) co-occur with catching SQLiteException. That means, we could infer that the method call Cursor.moveToFirst might cause SQLiteException. This is also confirmed by Android API documentation. Of course, calls to Integer.parseInt and Cursor.moveToFirst often also co-occur with catching Exception and Throwable, the most general exception in Java.

Now, assume that a programmer is writing a code snippet containing a call to Cursor.moveToFirst to read data from a database query and a call to Integer.parseInt to parse such data. The above exception handling rules learned from high quality code suggest

him/her to catch NumberFormatException and SQLiteException (or the more general ones Exception and Throwable). However, those rules are *imperfect* (e.g., recommending to catch NumberFormatException is applicable for only 32.6% of calls to Integer.parseInt).

In this paper, we address that issue via fuzzy logic. We developed XRank, a fuzzy logic system in FuzzyCatch to rank and recommend exception types. Basically, XRank contains a collection of rules *"if call m then catch e"*, in which m is an method (e.g., Integer.parseInt) and e is an exception type (e.g., NumberFormatException). Because those rules are imperfect, each rule has a confidence level $\mu_m(e)$.

XRank uses fuzzy sets to represent those rules. For each method m, X_m is defined as *"the set of exceptions to catch when calling m"*. As a fuzzy set, X_m has a membership function $\mu_m()$ which calculates membership score $\mu_m(e)$ for every available exception type e.

In traditional fuzzy logic systems, membership functions are defined by domain experts. However, in FuzzyCatch, we define membership functions empirically, i.e., the membership scores $\mu_m(e)$ is estimated from the training data (a repository of high-quality code). The key assumption is that the more m and e co-occur in high-quality code written by professional programmers, the more confidence that if we call m, we should catch e (because professional programmers often do that). Therefore, $\mu_m(e)$ is calculated based on the co-occurrence of m and e in the training data. Assume that n_m is the number of calls to m and $n_{m,e}$ is is the number of calls to m in all try-catch blocks catching e, then:

$$\mu_m(e) = \frac{n_{m,e}}{n_m} \tag{4.1}$$

Formula 4.1 implies that the value of $\mu_m(e)$ is between $[0, 1]$. Table 4.4 lists the top five exception types to catch for several methods and their membership scores estimated from our experiment data. We could see that for Integer.parseInt, NumberFormatException is the top-1 exception type, co-occurring with 32.6% of its method calls. The next one is the

Table 4.4: Membership scores

Integer.parseInt (n_m = 38,218)	$n_{m,e}$	$\mu_m(e)$
NumberFormatException	12,469	0.326
Exception	5,496	0.144
Throwable	687	0.018
IllegalArgumentException	470	0.012
JSONException	427	0.011
Cursor.moveToFirst (n_m = 23,406)	$n_{m,e}$	$\mu_m(e)$
SQLiteException	8,159	0.349
Exception	4,288	0.183
SQLException	460	0.020
Throwable	410	0.018
SQLiteFullException	216	0.009
MediaPlayer.getDuration (n_m = 1,152)	$n_{m,e}$	$\mu_m(e)$
IllegalStateException	105	0.091
Exception	44	0.038
Throwable	7	0.006
IllegalArgumentException	5	0.004
SecurityException	2	0.002
Byte.parseByte (n_m = 145)	$n_{m,e}$	$\mu_m(e)$
NumberFormatException	17	0.117
Exception	8	0.055
JSONException	3	0.020
Throwable	3	0.020
NoSuchElementException	2	0.013

general exception type Exception, co-occurring with 14.4%. The remaining exception types have much lower scores.

XRank utilizes the fuzzy sets X_m for two tasks: assessing if a given code snippet needs a try-catch block and ranking exception types to catch in that block. To do that, for each method m, it first calculates ρ_m, the *exception risk* of m, which is the total possibility that calling m would cause an exception. Because X_m is the fuzzy set representing the possibility that calling m would cause each exception e, the exception risk ρ_m is calculated as the total weight of X_m:

$$\rho_m = |X_m| = \sum_e \mu_m(e) \tag{4.2}$$

Table 4.5: Exception risk scores

Method	n_m	ρ_m
Cursor.moveToFirst	23,406	0.578
Integer.parseInt	38,218	0.512
MediaPlayer.getDuration	1,152	0.141

Table 4.5 lists the exception risk of three methods in Table 4.4. We can see that among the three of them, Cursor.moveToFirst is the riskiest, e.g., nearly 60% of its calls associated with catching an exception. In contrast, just 14% of calls to MediaPlayer.getDuration associate with catching an exception, thus, its exception risk is much lower.

Fuzzy logic systems often use linguistic concepts and variables to make them easier to use and understand for the domain experts and end-users of those systems. To make FuzzyCatch easier to use for programmers, we classify the methods into different risk classes based on their exception risk scores. Assume that α_1, α_2, and α_3 are respectively the 20%, 30%, and 50% percentile of all exception risk scores. Then, if method m has $\rho_m > \alpha_1$ (i.e., m is the top-20% among all methods), it is classified as "VERY RISKY". If $\alpha_1 \geq \rho_m > \alpha_2$ (m is not in top-20% but in top-30%), it is classified as "RISKY". And if If $\alpha_2 \geq \rho_m > \alpha_3$ (m is not in top-30% but in top-50%) it is classified as "LESS RISKY". The remaining methods are considered as not risky and ignored.

In the IDE (Android Studio), we color-code method calls with their risk levels. For example, calls of "VERY RISKY" methods like Cursor.moveToFirst are colored RED. Calls of "RISKY" methods (like Integer.parseInt) are colored ORANGE. Calls of "LESS RISKY" methods (like IntegerMediaPlayer.getDuration) are colored YELLOW. Calls of other methods are unchanged.

We could see that for a single method call to m, the exception risk ρ_m (which is correspondingly color-coded as visual guide for programmers) and the membership scores in the fuzzy set X_m are sufficient to suggest if a try-catch block is needed and to recommend exceptions to catch in such a block. For example, a call to Cursor.moveToFirst should be recommended with a try-catch block (because it is classified as "VERY RISKY" to cause an

exception). Then, the top exception to catch when calling Cursor.moveToFirst should be SQLiteException.

However, programmers often do not write try-catch block for each single call. We are more likely to write a try-catch block for a code snippet which might contain several method calls. Thus, to combine exception handling rules of all those method calls, XRank also uses a fuzzy logic approach.

Let us discuss an example first. Assume that a code snippet C contains a call to Cursor.moveToFirst to read data from a database query and a call to Integer.parseInt to parse such data. Then, because both of them have high exception risk, the code snippet also has high risk of causing exceptions. Two most potential exceptions are SQLiteException (from calling Cursor.moveToFirst and NumberFormatException (from calling Integer.parseInt). Thus, a programmer might write a try block around the code snippet and two catch blocks for those two exceptions. However, he/she might also write a simpler solution by using only a catch block catching Exception, which generalizes both NumberFormatException and SQLiteException and also has high membership scores in the fuzzy sets of exceptions of both Cursor.moveToFirst Integer.parseInt. In other words, when two methods Cursor.moveToFirst and Integer.parseInt are called together, the potential exceptions are combined from those for each of that method. In fuzzy logic, this combination is a fuzzy *union*.

In general, assume that C contains k method calls $m_1, m_2..., m_k$. Let X_C be the fuzzy set of exceptions should be caught for C, then X_C is the union of all fuzzy sets of exceptions for $m_1, m_2..., m_k$:

$$X_C = X_{m_1} \cup X_{m_1} \cup ... \cup X_{m_k} \tag{4.3}$$

Because X_C is a fuzzy set, it has a membership function μ_C. The union operation in fuzzy logic is defined via calculating μ_C from $\mu_{m_1}...\mu_{m_k}$. There are several formula for fuzzy union, we use the following one:

$$\mu_C(e) = 1 - (1 - \mu_{m_1}(e)) \times (1 - \mu_{m_2}(e)) \times ... \times (1 - \mu_{m_k}(e)) \tag{4.4}$$

Similarly, we also calculate the exception risk of the whole code snippet C as the following:

$$\rho_C = 1 - (1 - \rho_{m_1}) \times (1 - \rho_{m_2}) \times ... \times (1 - \rho_{m_k}) \tag{4.5}$$

For example, assume that e is Exception, m_1 is Cursor.moveToFirst, and m_2 is Integer.parseInt. From Table 4.4, $\mu_{m_1}(e) = 0.183, \mu_{m_2}(e) = 0.144$. Thus, $\mu_C(e) = 1 - (1 - 0.183)(1 - 0.144) = 0.301$. This membership score is higher suggesting that Exception is more likely to be used in the catch block when both methods are called. Similarly, from Table 4.5, $\rho_{m_1} = 0.578, \rho_{m_2} = 0.512$. Thus, $\rho_C = 1 - (1 - 0.578)(1 - 0.512) = 0.794$. This exception risk is higher suggesting that when both methods are called, it is higher possible that an exception will occur, which is intuitively plausible.

In summary, XRank recommends for a given code snippet C as the following. First, it calculates the exception risk ρ_C. If ρ_C exceeds the user preset risk level (e.g., VERY RISKY with $\rho_C > \alpha_1$), then it calculates $\mu_C(e)$ for every available exception type e and ranks them descendingly. FuzzyCatch presents the top exceptions in the ranked list to the programmer. When he/she selects one exception, FuzzyCatch will generate a try-catch block around code snippet C.

Improvements

To improve the prediction accuracy, we incorporate several programming language features to the XRank model. First, we observe that exceptions have hierarchical structure. For instance, in Java, all Exceptions and Errors are sub-classes of Throwable, all runtime exceptions are sub-classes of RuntimeException, and RuntimeException is a sub-class of Exception. With the hierarchical structure, programmers can use a super-class exception in any catch block of its sub-class exception. For example, in Figure 4.10, the programmer could use Exception in the catch block instead of its subclass RuntimeException. In XRank, we incorporate this hierarchical property by modifying the value of $n_{m,e}$ in the Equation

1. Anytime m appears in a try block while e appears in the corresponding catch block, in addition to increase the value of $n_{m,e}$ by 1, we also increase the value of $n_{m,e'}$ by a weighted value $w_{e'}$ if e' is a super-class of e. $w_{e'}$ could be calculated as the frequency of e' in all catch blocks:

$$w_{e'} = \frac{n_{e'}}{\sum n_e} \tag{4.6}$$

Most modern object-oriented programming languages support polymorphism. Overloading methods is one form of polymorphism in which methods within a class can have the same name if they have different parameter lists. For example, the Integer class has two methods for parsing numbers that have the same name parseInt(String s) and parseInt(String s, int radix). The later method has an additional parameter for radix. These overloading methods often perform similar tasks; thus, they are likely to throw similar exceptions. Furthermore, the later method in the example appears much less than the method parseInt(String s). Thus, learning rules for parseInt(String s, int radix) could be challenging due to lack of data. We improve the prediction accuracy of XRank by grouping overloading methods together and consider each group as one method when learning the model.

Finally, Table 4.4 also shows the fuzzy set of the API method Byte.parseByte. We can see that although parseByte is similar to parseInt and parseLong, the statistics of parseByte are small because it appears much less than the other methods. Thus, learning rules for parseByte could be challenging due to lack of data. To solve this problem, we group methods with the same aspect and learn the usage rules for the group. In our implementation, we manually identify 5 groups of similar API methods: number parsing (parseInt, parseLong,...), cursor reading (getInt, getLong,...), bitmap decoding(decodeByteArray, decodeResource,...), SQLiteDatabase (getVersion, getPath,...), text formatting (format, formatNumber, formatFileSize,...).

4.4.2 Recommending exception reaction

In addition to recommending what exception to catch, FuzzyCatch also recommends code to react when such an exception occurs by written in the catch block. Figures 4.10

```
//Usage 1
try {
  httpConnection.openConnection();
  httpConnection.setRequestProperty();
  ...
  httpConnection.connect();
} catch (IOException e){
  in = httpConnection.getErrorStream();
  ...
}
```

```
//Usage 2
try {
  httpConnection.getInputStream();
  httpConnection.getHeaderField();
  ...
} catch (IOException e){
  int responseCode =  httpConnection.getResponseCode();
  httpConnection.disconnect();
  ...
}
```

Figure 4.4: Handling exceptions for HttpURLConnection

and 4.11 illustrate code examples with different ways to write the code to react when an exception occurs. In Figure 4.10, the programmer adds the code that creates a new object with a different parameter. In Figure 4.11, the programmer adds the code to retrieve the error stream from the HttpURLConnection when the exception occurs. We consider the non-reaction in Figure 4.2 as a bad practice.

The reaction code written in the catch block can be large and complicated. However, Nguyen *et al.* [85] suggests that in most cases, programmers often call only one method for each object such as printStackTrace for an Exception object, close for an SQLiteDatabase object, or disconnect for an HttpURLConnection object. We consider creating a new object (such as Date now = new Date()) as calling a special method new. Similarly, assigning a value to an object (such as in = httpConnection.getErrorStream()) is like calling a special method assign.

Therefore, FuzzyCatch recommends code in the catch block by considering all objects appearing in the try block and providing a ranked list of methods to call for that object. XHand is the fuzzy logic system responsible to generate that ranked list.

One observation when developing XHand is that the method r to call for reaction depends on the method call and potentially cause the exception in the try block. For example, consider two usages of a HttpURLConnection in Figure 4.4. In the first usage, the exception may occur while using the HttpURLConnection for connecting to a server via connect method. In this situation, we would want to get the error stream by calling getErrorStream to get information about the error for further processing. In the second usage, the object is already connected to

the server but the read timeout expires causing an exception. In this situation, the exception handler, we would want to get the response code and disconnect from the server.

Thus, the key idea of XHand is to learn the fuzzy rule *"if call m in the* try *block then call r in the* catch *block to react or recover"*. For example, as in Figure 3, SQLiteDatabase.delete is called in the try block, thus, SQLiteDatabase.close is called in the catch block to release resources.

Thus, similar to XRank, we also define in XHand for each method m a fuzzy set R_m of methods to call for reaction to exception possibly caused by calling m. Its membership function ν_m is defined as:

$$\nu_m(r) = \frac{n_m(r)}{n_m} \tag{4.7}$$

In this formula, n_m is again the number of calls of m and $n_m(r)$ is the number of times m is called in a try block and r is called in the corresponding catch block.

Sometimes the method call for reaction depends on the occurring exception. For example, if the exception is Exception, then programmers usually call its method printStackTrace or getMessage. In other words, XHand also needs to learn the fuzzy rule *"if exception e occurs then call r to react or recover"*. Thus, for each exception e, we also define a fuzzy set R_e of methods to call for reaction if e occurs. Its membership function ν_e is defined as:

$$\nu_e(r) = \frac{n_e(r)}{n_e} \tag{4.8}$$

In this formula, n_e is the number of times e appears in a catch block m and $n_e(r)$ is the number of times r is called in that catch block.

Then, similar to XRank when a code snippet C calls k methods $m_1, m_2..., m_k$ and catches exception e. Let R_C be the fuzzy set of methods should be called for reaction, then R_C is the union of all fuzzy sets $R_{m_1}, R_{m_2}..., R_{m_k}$ and R_e. Its membership function is defined as:

$$\nu_C(r) = 1 - (1 - \nu_{m_1}(r)) \times (1 - \mu_{m_2}(r)) \times ... \times (1 - \nu_e(r)) \tag{4.9}$$

Finally, for each object o in C, if $r \in R_C$ is a method callable for o (e.g., r is a method of o, or r can accept o as an argument), it will be added to a list. The list is ranked descendingly by the membership scores and presented to the users of FuzzyCatch.

4.5 System implementation

In this section, we discuss the implementation and usage of FuzzyCatch. Currently available at rebrand.ly/exassist, it is released as a plugin of IntelliJ IDEA and Android Studio, two popular IDEs for Java programs and Android mobile apps.

4.5.1 Design overview

Figure 4.5 shows the design overview of FuzzyCatch. It has a) two modules to extract API usage models from source code and bytecode, b) one module to extract exception handling information from those usage models, c) two modules to train XRank and XHand from those extracted data, and d) two modules using the trained fuzzy logic systems to recommend code for catching and reacting to exceptions. Let us describe those modules in more detail.

GROUM extractor

FuzzyCatch employs an extension of GROUM (Graph-based Object Usage Model) presented in [9] to represent the raw API usages with exceptions. It has a module to extract GROUMs from the bytecode of Android apps in the training dataset. It also has a similar module to extract GROUMs from the code being written, which are used for its two tasks of recommending exception types and repairing actions. The extracting algorithms could be found at [83, 81, 77].

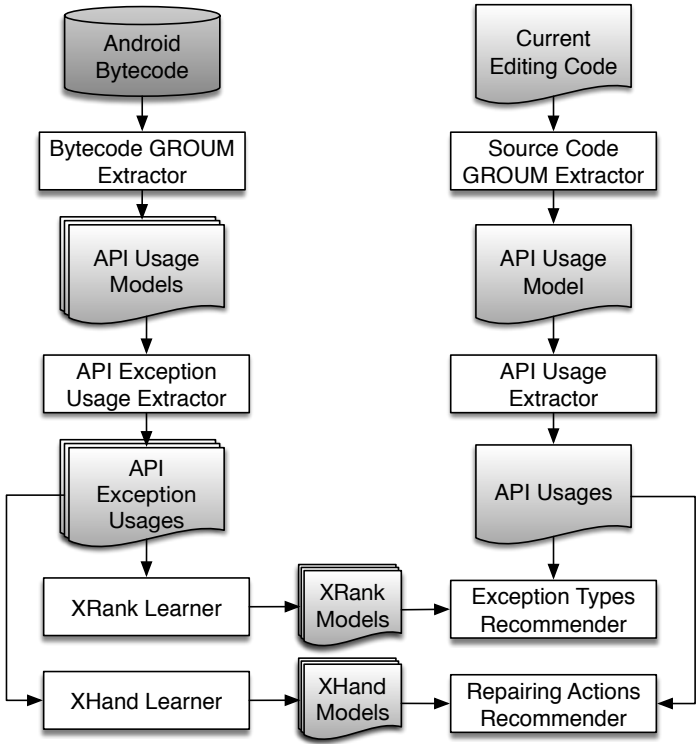

Figure 4.5: Design overview of FuzzyCatch

API exception usage extractor

Because XRank and XHand are trained from usages of API objects and method calls in exception handling code, FuzzyCatch has a module named API Exception Usage Extractor to extract from those usages from GROUMs. This module traverses each sub-graph of a GROUM representing a try-catch block and extracts all API method calls and the catching exception. The temporal order and data dependency between those API method calls are also extracted. The module then stores that information as an API exception usage.

XRank and XHand learners

These modules are responsible for training XRank and XHand systems from the extracted API exception usages. They count the raw occurrences n_m, $n_m(e)$, $n_m(r)$..., compute membership scores $\mu_m(e)$, $\nu_m(r)$... and exception risk scores ρ_m. (See Section 3 for details).

Exception types recommender

This module provides the recommendations on unchecked exception types for a selected code snippet C. It first extracts the set of API method calls that appeared in the under-editing code. Then, it utilizes XRank to compute exception risk scores for each call and make them color-coded in the IDE (e.g., red for VERY RISKY method calls). The programmer can use the color-code to select the code snippet of his/her interest for exception handling. Once the code is selected, XRank is used to calculate the membership scores of exceptions towards method calls appearing in the selected code. Top-ranked exceptions are added to the recommendation list.

Repairing actions recommender.

This module provides the recommendations for repairing actions in exception handling code. It extracts the set of API method calls in the try block the catching exception e. It then groups API method calls by objects. All API method calls in a group of an object have data dependency with that object. It uses the XHand system to recommend repairing actions for each object given the API method calls in the try block. Finally, it combines all the predict repairing actions to produce the final recommendation. Note that, FuzzyCatch converts special operations on objects like null check, assignments as pseudo method calls (e.g. Cursor.CHECK_NULL, String.ASSIGN_ZERO). When generating code, Cursor.CHECK_NULL is generated as: if(cursor!=null).

4.5.2 Usage

In this section, we present how to use FuzzyCatch in practice. Figure 4.6 shows a code example. Assume a programmer is writing code to open and get data from a database. She first opens a SQLiteDatabase and assigns it to variable sqliteDB. She then runs some SQL commands to update the database. Next, she creates a query that returns a Cursor object and uses that Cursor object to retrieve data to a list named bookTitles.

While the programmer is writing the code, FuzzyCatch utilizes XRank to access the exception risks and color-code its method calls. For example, the calls cursor.moveToFirst and cursor.moveToNext are red-flagged, bookTitles.add is orange-flagged. That makes the programmer aware that the code is dealing with database and calling Cursor's methods might cause *unchecked exceptions* (runtime exceptions) at runtime. (Without FuzzyCatch, the built-in exception checker in Android Studio only supports adding *checked exceptions*, thus, does not help her to make appropriate action in this case).

```
// create Cursor in order to parse our sqlite results
Cursor cursor = sqliteDB.rawQuery( sql: "SELECT bookTitle FROM " + tableName, selectionArgs: null);
// if Cursor is contains results
if (cursor != null) {
        // move cursor to first row
    if (cursor.moveToFirst()) {
        do {
            // Get version from Cursor
            String bookName = cursor.getString(cursor.getColumnIndex( columnName: "bookTitle"));
            // add the bookName into the bookTitles ArrayList
            bookTitles.add(bookName);
            // move to next row
        } while (cursor.moveToNext());
    }
}
Collections.sort(bookTitles);
```

Figure 4.6: Exception warnings by FuzzyCatch

Now, the programmer invokes FuzzyCatch by selecting the code snippet that she wants to check for exception then pressing Ctrl + Alt + R. In Figure 4.7, FuzzyCatch is invoked for a code snippet reading data from database with the Cursor object. It suggests a ranked list of unchecked exceptions with SQLiteException at the top. If the programmer chooses this exception, FuzzyCatch will wrap the currently selected code with a try-catch block with SQLiteException in the catch expression.

Figure 4.8 demonstrates the usage of FuzzyCatch in recommending exception repairing actions. After adding a try-catch block with SQLiteException for the code in the previous scenario, the programmer wants to perform recovery actions. To invoke FuzzyCatch for this task, she moves the cursor to the first line of the catch and presses Ctrl + Alt + H. In the example, FuzzyCatch detects that the Cursor object should be closed to release all of its resources and making it invalid for further usages. It also suggests to set bookTitles equals

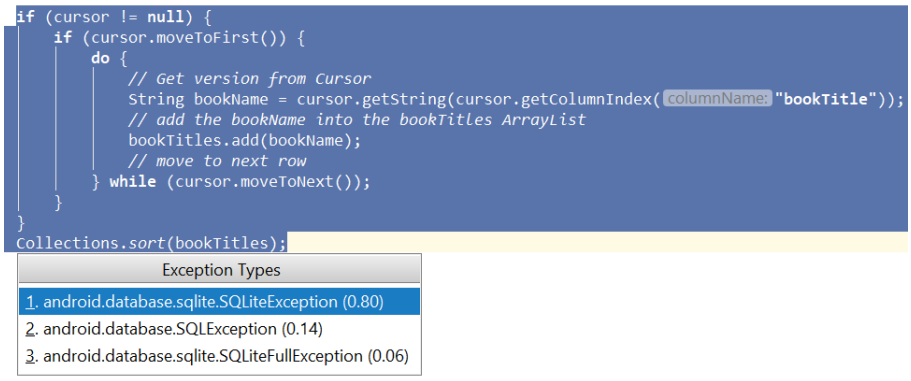

Figure 4.7: Recommending exception types

null to indicate the error while collecting data from cursor. If the programmer chooses the recommended actions, FuzzyCatch will generate the code in the catch block as in the figure. To save space, we show both the recommendation and generated code in a same window.

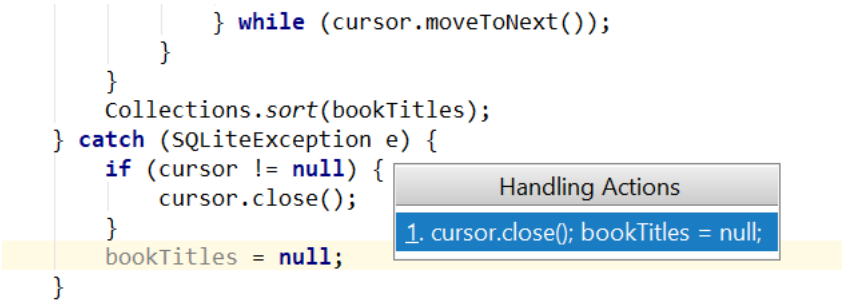

Figure 4.8: Recommending repairing actions

4.6 Empirical evaluation

We conducted several experiments to evaluate the effectiveness of FuzzyCatch on recommending exception handling code for Android and Java APIs. All experiments are executed on a computer running Windows 10 with Intel Core i7 3.6Ghz CPU, 8GB RAM, and 1TB HDD storage.

Table 4.6: Data collection

Number of apps	13,463
Number of classes	21,527,731
Number of methods	26,235,142
Number of bytecode instructions	741,912,624
Space for storing .dex files	165.0 GB
Number of exception types	261
Number of API methods	64,685

4.6.1 Data collection

The source code of most Android apps is not publicly available. With few apps with source code available, training FuzzyCatch from existing mobile app projects would be difficult and insufficient. Thus, we decided to train our models with Android bytecode. Table 4.6 summaries our dataset for training the models. In total, we downloaded and analyzed 13,463 top free apps from the Google Play Store. We fetched a list of top free apps from all the categories of the app store. Then, we only downloaded apps with an overall rating of at least 3 (out of 5), based on the assumption that the high-rating apps would have high code quality, and thus, would have better exception handling code. Since Android mobile apps are distributed as .apk files, our crawler unpacked each .apk file and kept only its .dex file, which contains the bytecode of the app. The total storage space for the .dex files of the downloaded apps are around 165 GB. After parsing those .dex files, we obtained about 21 million classes.

Next, we developed a bytecode analyzer that analyzed each class and looked for all methods in the class to build GROUM models. Since an Android mobile app is self-contained, its .dex files contain the bytecode of all external libraries it uses. That leads to the duplication of the bytecode of shared libraries. To remove that duplication, our bytecode analyzer maintains a dictionary of the analyzed methods and analyzed each of them only once. In the end, we analyzed over 26 million *unique* methods which have in total nearly 740 million bytecode instructions. They are used to train XRank and XHand, producing fuzzy logic rules for 261 exception types and 64,685 methods in Java and Android APIs.

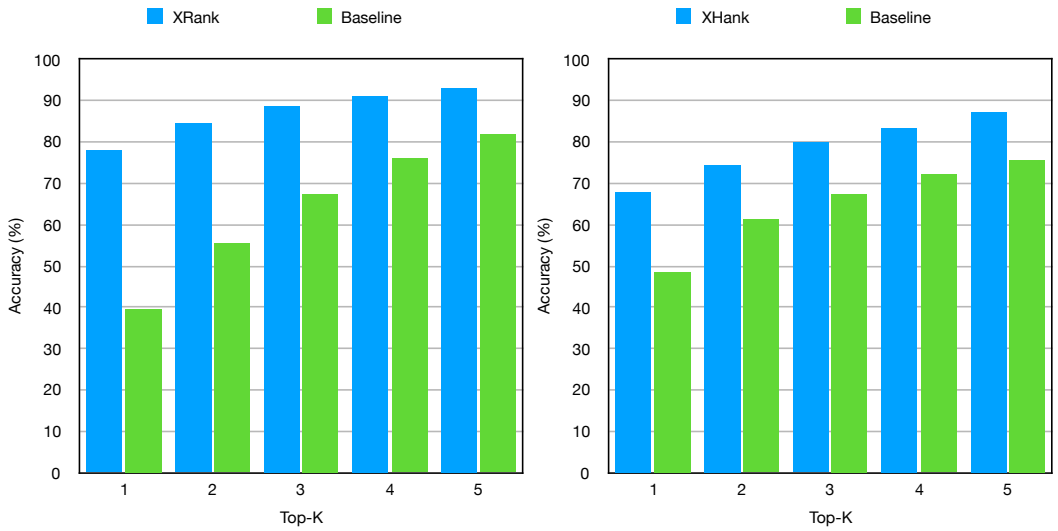

Figure 4.9: Recommendation accuracy

4.6.2 Experiments on Android bytecode

We consider the bytecode of the downloaded apps as the evaluation dataset and use 10-fold cross-validation to evaluate the accuracy of FuzzyCatch on them. We divide them into 10 folds, each contains roughly 1400 apps. At each iteration, we choose a fold for testing and the remaining for training. The final results are averaged from 10 iterations.

For each try-catch block in the code of the testing apps, we evaluated if FuzzyCatch can recommend the actual exception and reaction method calls in the catch block given the method calls in the try block. For example, if the code actually catches e and e is also in the top-k list recommended by FuzzyCatch then it is an accurate recommendation for catching exceptions. Similarly, if for an object o, the code calls method r for recovery, and r is also in the top-k recommended by FuzzyCatch then it is an accurate recommendation for reaction to exceptions.

To measure the effectiveness of FuzzyCatch, we compared it with a simple baseline. Because no other approaches have been proposed for this recommendation problem, we design the baseline as a simple approach based on the frequency of catching exception types and reaction calls. That is, we count the number of times an exception (or a repair call) appears in all try-catch blocks in the training data. Then, the exceptions (or repair calls) are ranked

descendingly by their frequencies and recommended as such. For example, Exception is the top-1 exception recommended by this approach. And close is the top-1 method recommended to call for repairing (or recover) a Cursor object.

Figure 4.9 shows the experiment results (the left chart for XRank and the right chart for XHand). From the figure, we can see that both XRank and XHand have consistently high levels of accuracy. For example, XRank has top-1 accuracy of 77% and top-5 accuracy of 92%. For XHand, they are 69% and 87%. Also, both models significantly outperform the baselines.

4.6.3 Experiments on real exception bugs

We conducted two more experiments to evaluate on the effectiveness of FuzzyCatch on detecting and fixing real exception bugs.

Data collection

In this section, we describe briefly our dataset of exception handling bugs and fixes. We focus on bugs that are caused by not catching exceptions in the code. For convenience, we defined those bugs as **exception bugs**. The fix for those types of bugs is called **exception bug fixes**. We first collected a dataset consists of several large open-source Android projects. For each project, we checked out its source code repository to retrieve all the code and commits. We developed an extraction tool to identify the bug fixes from the commits and issues of those projects. Next, we manually inspected all the bug fixes to identify exception bug fixes. Finally, we were able to collect a dataset of 1,000 exception bug fixes. The steps to identify the exception bug fixes and collect the dataset are described in detail in [85]. Figure 4.10 shows an exception bug fix in our dataset. The dataset is available at rebrand.ly/ExDataset.

COMMIT MESSAGE: "fix #2695: re-introduce a workaround we were using in previous versions"

```
- postContent = new SpannableStringBuilder(
-                                 mEditorFragment.getSpannedContent());
+ try {
+     postContent = new SpannableStringBuilder(
+                                 mEditorFragment.getSpannedContent());
+ } catch (RuntimeException e) {
+     // A core android bug might cause an out of
+     // bounds exception, if so we'll just use the current editable
+     // See https://code.google.com/p/android/issues/detail?id=5164
+     postContent = new SpannableStringBuilder(
+         StringUtils.notNullStr((String) mEditorFragment.getContent()));
+ }
```

Figure 4.10: An exception bug fix

Detecting exception bugs

By calculating exception risk scores of method calls and code snippets, FuzzyCatch could warn programmers of missing exception handling code, i.e., it could detect exception related bugs. For example, if a method call having a "VERY RISKY" exception risk score is not covered by a try catch block, FuzzyCatch can warn the programmer of a potential exception bug and recommends an exception to catch.

To evaluate FuzzyCatch on such warnings, we manually applied FuzzyCatch for the 1,000 exception bugs that we collected. We run FuzzyCatch on the buggy code version and check if FuzzyCatch produces warnings. Then, we compare the exception recommended by FuzzyCatch and the actual exception caught in the fixed version. The three risk levels are defined the same as in Section 3.1.

Table 4.7 lists the results. If we set FuzzyCatch to the strictest mode, i.e., only warning on method calls with 'VERY RISKY' exception risk level then it produced warnings for 734 out of 1,000 bugs (73.4%). In the loosest mode, i.e., warning on all method calls with 'LESS RISKY' exception risk level and higher, then it produced warnings for 821 out of 1,000 bugs (82.1%).

The prior experiment on Android bytecode shows that FuzzyCatch does not always recommend the exception actual used by the programmers. For example, the top-1 recommendation for Integer.parseInt is NumberFormatException. However, in many cases the programmers actually use the more general Exception. Thus, it is similarly in this experiment.

In 734 cases that FuzzyCatch warns of potential exception bugs in its strictest mode, the top-1 recommended exception is actually used in 547 cases (74.5%).

When using FuzzyCatch, there could be a case in which the tool flags a warning on a method that the programmer does not want to add try-catch block or catch an exception. We defined those cases as "unwanted warnings". To evaluate if FuzzyCatch produces unwanted warnings, we selected 100 code snippets from the latest version of our subject systems. We only selected code snippets of at least 10 lines of code and having no try catch block added through the whole project history. We assumed that such code snippets will not need exception handling code. Thus, if FuzzyCatch produces a warning (of missing exception handling) for such a code snippet, it will be an unwanted warning. In a real-world, large-scale system, it is nearly impossible to prove or verify that a piece of code C is free of bugs. So we did not have a reliable method to determine if C does not need a try-catch statement. We instead looked at its revision history. Because the subject systems have a long history (some have up to 28,000 revisions), if a piece of code has not been enclosed by a try-catch statement throughout such long history then we could reasonably assume that it does not need.

In the strictest mode, FuzzyCatch produced 8 unwanted warnings (out of 100 code snippets). In the loosest mode, FuzzyCatch produced 35 unwanted warnings. Thus, this is a balanced trade-off. If the programmer wants to higher code quality (i.e., fewer exception bugs), he needs to spend more time on FuzzyCatch warnings to determine if its warnings and recommendations are necessary.

In addition, adding a try-catch statement makes the code safer. The expected loss for an "unwanted warning" is much smaller than a bug caused by a missing try-catch statement. Thus, we conservatively call recommendations of FuzzyCatch "unwanted warnings" because programmers have not yet wanted to use them rather than "false positives" meaning those recommendations are strictly wrong.

Table 4.7: Recommendation results for 1,000 exception bugs

Risk level	VERY RISKY	RISKY	LESS RISKY
Correct warning	734	783	821
Top-1 Accuracy	547 (75%)	587 (75%)	601 (73%)
Top-3 Accuracy	593 (81%)	618 (79%)	646 (80%)

Table 4.8: Results in recommending repairing actions

	FuzzyCatch	Barbosa *et al.*	CarMiner
# of fixes	437	437	437
# of matches	287 (65%)	126 (28%)	196 (44%)

Repairing exception bugs

In this experiment, we evaluate the effectiveness of FuzzyCatch in recommending repairing actions for real exception bug fixes. Not all of the exception bug fixes contain handling actions, for example, the fixer could add code to swallow exceptions, add logging messages, or re-throw another exception. We select a subset of the 1,000 exception bug fixes which includes all the exception bug fixes that contain handling actions by programmers in the catch block. In each bug fix of the subset, developers performed at least one repairing action (i.e. a method call, an assignment, etc.) in the exception handling code. Figure 4.11 shows an example of a bug fix in the subset. In the bug fix, the developer performed three method calls and one assignment to set value for postContent. In total, there are 437 out of 1,000 exception bug fixes have handling actions, thus, we evaluate FuzzyCatch on this subset.

For each bug fix in the dataset, we invoked FuzzyCatch to recommend repairing actions and compare the recommendation result with the corresponding fix. If the recovery actions recommended by FuzzyCatch exactly match with the fix, we count as a match. Otherwise, we consider the case as a miss.

To further evaluate the effectiveness of FuzzyCatch in this task, we compare our tool with two existing approaches presented in [14, 103]. Barbosa *et al.* [14] proposed a

technique that uses exception types, method calls, and object types as heuristic strategies to identify and recommend relevant code examples with current handling code under editing. Although the approach does not provide actual recommendations, the examples that it suggests could be used as a guide for programmers to fix an exception bug. Thus, we evaluate the baseline as follows: if at least one of the top-5 relevant code examples recommended by the baseline matches with the fix, we count as a match, otherwise, we consider the case as a miss. We re-implemented the technique using the same configurations presented in the work. As the approach requires a corpus of real code examples, not bytecode, we implemented it with a dataset contains the source code of 1,514 Android projects listed on the FDroid app repository [2]. The list of all the projects in the dataset could be found at rebrand.ly/ExDataset. Rahman *et al.* [91] proposed another approach that recommends exception handling code examples from a number of GitHub projects. The method is difficult to implement as it requires to collect 9 complex code features from each code example and to estimate 12 weighting parameters using machine learning. We contacted the author but we could not compile their code or collect their training and experiment data. Thus, we did not compare the result with their work.

We also compare our method with CAR-miner [103], an approach that uses association mining techniques to mine association rules between method calls of `try` and `catch` blocks in exception handling code. The model was used to detect bugs related to exceptions. The mined rules have a form (S_n, S_e) in which S_n is the set of all methods in a `try` block while S_e is the set of repairing methods in a `catch` block. S_e could be used as the recommendation on handling actions. We re-implement CAR-miner with the same settings that were used in the original paper on the same Android bytecode dataset as FuzzyCatch.

Table 4.8 shows the evaluation result of FuzzyCatch and the baseline for the task. In a total of 437 exception bug fixes, the recommendations of FuzzyCatch match the fixes of developers in 287 cases. Overall, FuzzyCatch could provide meaningful recommendations

[2]https://f-droid.org/

in roughly 65% of the bug fixes. The first baseline only matches the fixes of developers in 126 (28%) cases. The association mining technique CAR-miner performs better with 196 cases (44%). Overall, the result of FuzzyCatch outperforms the baseline methods substantially.

COMMIT MESSAGE: "Merge pull request #937 from garvankeeley/bug-network"

```
- InputStream in = new BufferedInputStream(
-                       httpURLConnection.getInputStream());
  InputStream in = null;
+ try {
+    in = new BufferedInputStream(
+                       httpURLConnection.getInputStream());
+ } catch (Exception ex) {
+    in = httpURLConnection.getErrorStream();
+ }
```

Figure 4.11: An example of handling an exception

For demonstration, Figure 4.11 shows an exception bug fix in which the three techniques have different recommendation results. In this example, FuzzyCatch recommends calling getErrorStream on httpURLConnection object and an assignment for the InputStream object. The result exactly matches the code of the programmer as shown in the figure. CAR-Miner mined (getInputStream, disconnect) as a association rule, thus, it suggests calling the method disconnect instead of getErrorStream, which is incorrect. The code examples provided by the first baseline did not have any meaningful recommendations.

4.7 Discussion

In this section, we discuss several aspects of FuzzyCatch in more detail. From machine learning perspective, FuzzyCatch is a simple ensemble approach to three classification problems: Given code snippet C, (1) predict if it needs an enclosing try-catch, which is a binary classification problem; (2) predict what exception to catch, which is a multi-class prediction problem with classes are all exception types; (3) predict what method to call in the catch block, which is a multi-class predict problem with classes are all API methods. FuzzyCatch learns and predicts like Naive Bayes [37] but more flexibly. Because fuzzy

membership functions are not probability distribution functions, FuzzyCatch does not need to assume the independence of predictors or normalize $\sum_e \mu_C(e) = 1$.

We could consider FuzzyCatch is a big code powered fuzzy logic system specially designed for the software engineering domain. It represents exception handling knowledge as fuzzy logic rules. It uses fuzzy set theory to model and apply those rules, and uses fuzzy union operations to combine those rules. In the traditional fuzzy logic system, variables are often continuous such as Temperature, Density or linguistic like LOW, VERY LOW. The membership functions are often manually defined by domain experts with functions such as triangular or trapezoidal. In FuzzyCatch, variables are discrete, i.e. methods, exceptions, and the membership functions are estimated automatically from millions of code examples.

FuzzyCatch is trained from a dataset contains over 13,000 highly rated Android apps. We assume that FuzzyCatch can learn common, statistically significant programming patterns from its huge collection of code. Also, it is reasonable to believe that the dataset contains high-quality code, thus, better exception handling code, because it contains apps that are developed and frequently updated by top software companies, e.g. Google, Facebook, which employs the best software engineers in the world. They are used daily by millions to billions of users worldwide, any errors are likely discovered and reported quickly. Our empirical evaluation also confirms that FuzzyCatch can be used effectively to detect and fix exception-handling bugs.

4.8 Threats to validity

The threat to internal validity includes errors when we identified and evaluated exception bug fixes. Firstly, we might incorrectly identify bug fixes. To reduce this threat, we could apply more patterns (e.g. adding more filter keywords) when identifying bug fixes from commits. Secondly, we might incorrectly identify exception bug fixes from the bug fixes. As the evaluation process is manually carried out by three researchers, there are might be errors

when reporting results due to human errors. The threat could be reduced by adding more people to our labelling and evaluation process.

The threat of external validity includes our selected projects for labeling exception bugs and fixes. Although we analyzed medium to large Android projects, the number of exception bugs and fixes from the selected projects may still be limited. To reduce this threat, more exceptions bugs and fixes shall be collected to increase the reliability of our results. In our evaluation, other measures such as precision and recall should be introduced. We could not compute such measures due to the fact that we do not have "negative" examples in the test set, i.e. the code examples that are not needed to catch exceptions. Given a piece of code, it is impossible to determine that the code will never throws an exception. Thus, we could not collection these 'negative" examples.

Chapter 5

Personalized Code Recommendation Model

5.1 Introduction

In software development, programmers must interact with large amounts of different types of information and perform many activities to build an application. They constantly need to figure out which variables, objects, or methods to use next. Additionally, the number of objects and methods to use in the current project or libraries are often huge, which makes programmers impossible to remember all the usages. To help programmers work more effectively, modern integrated development environments (IDEs) offer code recommendation features. These tools help developers to complete the names of classes, methods, fields, and keywords. Murphy *et al.* [69] performed a study indicates that programmers could use the code recommender up to several times per minute when they develop applications in Eclipse.

However, the default code recommendation plugins inside current IDEs offer fairly limited functionalities. Firstly, the current recommendation tools often provide the ranking candidates based on alphabetical order. Certain candidates have a higher probability to appear than others but might not be included at the top of the ranked list. The recommendation could be time-consuming if the number of candidates is big, and the user needs to move down the rank list to find what he wants. Secondly, the built-in tools often lack the consideration of code context when making a recommendation. For example, let assume a user created a new URL object in the previous line, he is likely to create a HttpURLConnection object by calling the openConnection method on the newly created URL object. Thus, a tool should recognize the existence of the URL object as the context when making recommendations.

To further improve the effectiveness and usefulness of current code recommendation tools, multiple methods have been proposed [16, 83, 92, 70, 116]. Most of the techniques are

motivated by the crowd-based approach. The approach focuses on the common code patterns of objects and methods that are shared among multiple programmers. The idea is to build a large dataset by collecting a large pool of available source code. Next, common code patterns are extracted or inferred from the dataset. In the recommendation phase, the current code context is used to match with learned code patterns to infer the recommendations.

At the same time, each programmer has certain coding preferences and styles. For example, a programmer could prefer to use CSVReader object to read a file, while others prefer to use BufferedReader. These different coding preferences are referred to as personal coding patterns of programmers. In the crowd-based approach, while common code patterns are combined and inferred, such coding preferences are blurred. This could limit the accuracy of the code recommendation tool for a specific programmer. To capture the personal coding patterns, a code recommendation tool should take into consideration the code history of written the programmer. For example, which classes, objects, or code patterns that the programmer often uses. Providing such personal recommendations could improve the effectiveness and enhance the user satisfaction of the tool. Our preliminary study [73] shows a recommendation model that incorporates personal code patterns provides improvements in suggesting variable declaration and initialization code. Therefore, it is desirable to combine both personal and common code patterns to improve current code recommendation models.

We propose Persona, a novel code recommendation model that focuses on the personal coding patterns of programmers while also combines with project-specific and common code patterns. As a personalized model, Persona is built and updated for each programmer. It is composed of three sub-models: PerCR, a model that captures personal code patterns of a programmer; ProCR, a model that captures the project-level code patterns that the programmer is working on; and GenCR, a general model that capture code patterns shared between multiple projects. Persona incorporates code patterns learned from the three sub-models together and utilizes those patterns for recommending code elements including variable names, class names, methods, and parameters.

PERSONA utilizes the fuzzy set theory [55] to model correlation/association between code elements. It defines a fuzzy set of potential recommendation candidates toward code elements that appear in the current code context. Each candidate has a membership score, which determines a certain degree of membership in the fuzzy set. The membership score is calculated based on various factors such as the code history of the programmer, the project he is working on, or common code patterns. The candidate with a higher membership score will be ranked higher in the recommendation list.

To build the proposed recommendation model, we extract personalized object usage instances from the code history of a programmer. We use such data to train a personalized code recommendation model PERCR for the programmer. The code history of other programmers in the current project is also extracted to train a project-level recommendation model PROCR. We also train GENCR, a general model to capture common code patterns on a large code corpus. Finally, we incorporate the sub-models together to build PERSONA. Once trained, given the current editing code in which the programmer wants to invoke code recommendation, our recommendation tool extracts its context features and utilizes PERSONA to compute the recommendation rank list.

We have conducted several evaluation experiments to evaluate the usefulness and effectiveness of the personalized code recommendation approach. In the evaluation, PERSONA is trained on a big dataset, containing 14,807 Java projects across multiple domains, amounting to over 350 million lines of code in over 2 million files. Next, the model is evaluated on 10 large Java projects with the number of commits in each project is ranging from 23,000 to over 400,000. The evaluation results show that PERSONA could achieve high accuracy in code recommendation. For example, when evaluating on a programmer, our approach has top-1 accuracy of 66% and top-3 accuracy of 74%. Furthermore, our model also outperforms the baselines significantly in top-1 accuracy in these experiments. It outperforms the first baseline by an average of 12-15% and generates a gap of 4-6% when compared to the second baseline. We also show that the recommendation accuracy of PERSONA improves over time

as more code of the programmer is used to train. By incorporating three sub-models together, the PERSONA performs reasonably well even if the code history of the programmers is thin in the project.

5.2 Motivation

Let us start with an example that explains the challenges when using the current code recommendation methods and motivates our approach. Fig 5.1 shows a code recommendation scenario in which the programmer writes code to read a file. In the first line, he creates an InputStream object from the filename. Next, he creates an InputStreamReader object from the InputStream. Let us assume that he invokes code recommendation at the first of line 4. A code recommendation method based on the crowd-based approach would recommend creating a BufferedReader (line 5). The reason is using BufferedReader to read file from InputStreamReader is a common code pattern that is often shared between programmers. It learns the pattern from mining a code corpus.

```
1. // Load file
2. InputStream is = testContext.getAssets().open(filename);
3. InputStreamReader inputStreamReader = new InputStreamReader(is);
4. _
5. // BufferedReader bf = new BufferedReader(inputStreamReader);
6. // CSVReader csvReader = new CSVReader(reader);
```

Figure 5.1: A code recommendation scenario

The preference of the programmer in the example is different. He prefers to use CSVReader object to read file instead of BufferedReader. He has been using CSVReader throughout his application development. Thus, an ideal code recommendation tool should prioritize the personal code patterns and recommends CSVReader (line 6).

The example shows that programmers have preferences and styles when coding including naming variables, using certain classes, objects, and methods, or applying certain coding patterns. Thus, such personal preferences should be taken into consideration when providing

code recommendations as it could improve the effectiveness and enhance user satisfaction of the recommendation tool.

5.3 Model

In PERSONA, code recommendation is modeled as a ranking problem: given the current editing code E in which a programmer is asking recommendations for a missing code element, Φ is the set of all possible recommendation candidates, find a candidate $c \in \Phi$ with the highest possibility to be filled in the current missing location.

The key idea of PERSONA is to rank potential candidates c toward a set of context features F in E by modeling the correlation/association of c with each context feature in F. The set of feature F includes object types, method calls, variable names, and parameters that occur in E. If a candidate c has a higher correlation with features in F, c is considered to have a higher possibility and will be rank higher in the list.

For example, in Fig 5.1, in which the programmer invokes code recommendation at beginning of line 4. The goal of PERSONA is to rank CSVReader as the declaration type with highest possibility. The set of context features F includes object types {InputStream, InputStreamReader}, variables {is, testContext, inputStreamReader}, method calls {getAssets, open, InputStreamReader.new }, and parameters {fileName}.

To model correlation/association between candidates and context features, PERSONA utilizes the fuzzy set theory [55]. It defines a fuzzy set of potential candidates toward a context feature as follows.

Definition 1 (Potential candidate) *For a specific context feature f, a fuzzy set C_f, with an associated membership function $\mu_f()$, represents the set of potential candidates toward f, i.e. candidates that are highly correlated with f*

Fuzzy set C_f is determined via a membership function $\mu_f()$ with values in the range $[0, 1]$. For a candidate c, the membership score $\mu_f(c)$ determines the certainty degree of the

membership of c in C_f, i.e. how likely does c belong to the fuzzy set C_f. $\mu_f(c)$ represents the degree of association between c and f. $\mu_f(c)$ also determines the ranking of c toward f. If $\mu_f(c) > \mu_f(c')$ then c is considered higher correlated to f to c'. The membership score is often computed as follows.

Definition 2 (Membership score) *The membership score $\mu_f(c)$ is computed as the correlation between the set D_f representing usages of the context feature f, and the set D_c representing usages of the candidate c:*

$$\mu_f(c) = \frac{|D_f \cap D_c|}{|D_f \cup D_c|} = \frac{n_{f,c}}{n_f + n_c - n_{f,c}} \tag{5.1}$$

where, n_f is the number of usages of the context feature f, n_c is the number of usages of the candidate c, and $n_{f,c}$ is the number of times that the candidate c co-occurs with f. As the Equation 5.1, the value of $\mu_f(c)$ is between $[0, 1]$. If $\mu_f(c) = 1$, then c always occurs on the code snippets that contain f, thus, given a code snippet contains f, it is very likely that c co-occurs. If $\mu_f(c) = 0$, it means that c never occurs on code snippets that contains f, thus, given a code snippet contains f, it is unlikely to recommend c. In general, the more frequently c co-occurs with f, the higher value of $\mu_f(c)$.

Based on the fuzzy logic framework described above, we develop three different code recommendation models. Each model has its membership score function (Equation 5.1) and is learned from different datasets. Finally, we incorporate the three models together to build PERSONA. Let us describe each model as follows.

5.3.1 Personalized code recommendation model

As demonstrated Section 3, programmers have different coding preferences, styles, experience levels, and knowledge about libraries and frameworks. For example, a programmer might prefer using certain classes or methods than others; some programmers prefer short variable names for a BufferedReader object such as b or bf, while others use long names such

as bufferedReader, etc. In other words, there are personal code patterns that appear in the code written by a programmer. Thus, a code recommendation model that utilizes those personal code patterns could improve the code recommendation performance significantly. Based on this observation, we design a personalized fuzzy-based code recommendation model (or PERCR for short).

Let us assume a programmer d is working on a project P, H_d is the code history written by the programmer in the current project. The membership score in PERCR is defined as:

$$\mu_d(f, c, H_d) = \frac{n_{f,c}(H_d)}{n_f(H_d) + n_c(H_d)} \tag{5.2}$$

where $\mu_d(f, c, H_d)$ represents the membership score of candidate c in the fuzzy set C_f of the context feature f, $n_f(H_d)$ represents the usages of f in H_d, $n_c(H_d)$ represents the usages of c in H_d, and $n_{f,c}(H_d)$ represents the usages in H_d which f and c co-occur. Normally, $n_f(H_d)$ is defined as the number of occurrences of f in H_d. However, in PERCR we also want to model the change in code patterns of programmers over time. For example, a programmer might start by using the BufferedReader object to read file but as he writes more code, he gradually changes his preference to using CSVReader. Thus, in PERCR, $n_f(H_d)$ is computed as follows:

$$n_f(H_d) = \sum_{x \in H_d : f} e^{-\frac{1}{\Delta t_x}}$$

where $\Delta t_x = t_x - t_0$ is the time decay, t_0 is the timestamp in which the project start, t_x is the timestamp in which f occurs in H_d. The idea behind the formula is that the occurrence of f later in the project has more influence over the previous occurrences. Similarly, we have:

$$n_c(H_d) = \sum_{x \in H_d : c} e^{-\frac{1}{\Delta t_x}} \text{ and } n_{f,c}(H_d) = \sum_{x \in H_d : f,c} e^{-\frac{1}{\Delta t_x}}$$

5.3.2 Project-level code recommendation model

When multiple programmers work on the same project, they read, share, and reuse the code of each other. Thus, the code written by a programmer could be influenced by other programmers in the same project. For example, a programmer could create and use MapUtil class that contains several utility methods for Map. Other programmers in the same project also reuse the class. Thus, the code patterns related to the class could be shared between programmers in the project. We present ProCR, a fuzzy-based model that captures the project-level code patterns in the project that the programmer is working on.

Let us assume a programmer d is working on a project P, and $P - H_d$ is the code history written by all other programmers (except d) in the project. ProCR is the project-level code recommendation model defined specifically for the programmer d. The membership score in ProCR is defined as:

$$\mu_d(f, c, P - H_d) = \frac{n_{f,c}(P - H_d)}{n_f(P - H_d) + n_c(P - H_d)} \tag{5.3}$$

where $\mu_d(f, c, P - H_d)$ represents the membership score of candidate c in the fuzzy set C_f of the context feature f. Other terms in Equation 5.3 are defined similarly to corresponded terms in Equation 5.2. In other words, the project-level model ProCR is defined similarly to the personalized model PerCR. The difference is that PerCR is trained from the code history H_d of the programmer d, while ProCR is trained on the code history of other programmers in the same project.

5.3.3 General code recommendation model

In modern application development, programmers rely heavily on shared APIs to write code. For example, two different programmers could use the same API classes such as BufferedReader, File to read data from a file. The usage pattern of using those objects could be similar between the two programmers. Programmers might also share programming

conventions of programming languages such as naming conventions. Thus, programmers do share common code patterns and we want to incorporate these patterns to our approach to improve the recommendation accuracy. We propose GenCR, a fuzzy-based model that captures such common code patterns shared between multiple projects. The membership score of GenCR is defined as follows:

$$\mu(f, c, \overline{P}) = \frac{n_{f,c}(\overline{P})}{n_f(\overline{P}) + n_c(\overline{P}) - n_{f,c}(\overline{P})} \tag{5.4}$$

where $\overline{P}$ is the set contains the code of all projects in the dataset except the current project P, $n_f(\overline{P})$ is the number of occurrences of f in $\overline{P}$, $n_c(\overline{P})$ is the number of occurrences of c in $\overline{P}$, and $n_{f,c}(\overline{P})$ is the number of times that the candidate c co-occurs with f in $\overline{P}$.

5.3.4 Combining sub-models

Using each sub-model described above separately could yield a low-accuracy recommendation. For example, if a programmer just joined the project or the project just started, there is not much data to train PerCR and ProCR. Thus, these models could be fairly inaccurate, while GenCR could not recommend personal or project-level code patterns. To maximize the recommendation accuracy, we design Persona to incorporate the three sub-models together. It defines the membership score $\mu_f(c)$ in Persona is as follows:

$$\mu_d(f, c) = \alpha_1 \mu_d(f, c, H_d) + \alpha_2 \mu_d(f, c, P - H_d) + \alpha_3 \mu(f, c, \overline{P}) \tag{5.5}$$

where $\alpha_1 + \alpha_2 + \alpha_3 = 1$ are weighting coefficients. The value of α_i represents the contribution level of a sub-model towards Persona, the higher value of α_i the bigger contribution of the sub-model. If the model defines the membership score $\mu_f(c)$ using Equation 5, we call the model PersonaSum.

As the sub-models are defined in separated datasets, the membership score of Persona could also be defined using the max function:

$$\mu_d(f, c) = \max(\mu_d(f, c, H_d), \mu_d(f, c, P - H_d), \mu(f, c, \overline{P})) \tag{5.6}$$

In Equation 5.6, the sub-model with the highest value of membership score will decide the value of $\mu_d(f, c)$. If the model defines the membership score using Equation 6, we call the model PersonaMax. We experimented both approaches of calculating $\mu_d(f, c)$ in Persona in our evaluation.

After defining the membership score function, we show how Persona calculates the rank list of candidates using the fuzzy set theory. Based on the definition of potential candidates toward a context feature f as a fuzzy set (Definition 1), Persona defines potential candidates toward a set of context features F using the union operation of fuzzy set theory as follows.

Definition 3 *Given a set of context features F, a fuzzy set C_F, with an associated membership function $\mu_F()$, represents the set of potential candidates toward F, i.e. the candidates that are highly correlated with context features of F. C_F is computed as the union of the fuzzy sets C_f of context features in F:*

$$C_F = \bigcup_{f \in F} C_f \tag{5.7}$$

Because C_F is a fuzzy set, it has a membership function μ_F. The union operation in fuzzy logic is defined via calculating μ_F from $\mu_{f_1} ... \mu_{f_k}$. There are several equations for fuzzy union operation, we use the following one:

Definition 4 *The membership score $\mu_F(c)$ is calculated as the combination of the membership scores $\mu_f(c)$ of its associated context feature f:*

$$\mu_F(c) = 1 - \prod_{f \in F} (1 - \mu_f(c)) \tag{5.8}$$

In Equation 5.8, $\mu_F(c)$ represents the correlation of candidate c toward a set of context features F. As the equation, we see that the value of $\mu_F(c)$ is also between $[0, 1]$ and represents the likelihood in which the candidate c belongs to the fuzzy set C_F, i.e. the set of potential candidates for the set of context features F. $\mu_F(c) = 0$ when all $\mu_f(c) = 0$, which means that c never occurs on any code contains a context feature in F. Thus, Persona considers that c

is unlikely to occur on the code contains F. If there is any method f is F with $\mu_f(c) = 1$, then $\mu_F(c) = 1$, or PERSONA considers that c is very likely to occur on the code contains F as c always occurs on code contains f in F. In general, the more context features f in F with high $\mu_f(c)$ values, the higher $\mu_F(c)$ is, or c is more likely to occur on the code contains F.

In the code recommendation phase, PERSONA ranks candidates based on the value of $\mu_F(c)$ and provides the rank list for the user. The higher value of $\mu_F(c)$, the higher ranking of the candidate c in the list.

5.4 System implementation

5.4.1 Overview

In this section, we briefly discuss the points in the design and implementation of our recommendation system. Fig 5.2 shows an overview of the system. Overall, it consists of 3 main components. The code history extractor is the component for extracting personalized object usage instances from the code history of a programmer. The model learner uses the extracted data to train and incorporate the three sub-models in PERSONA. Finally, the code recommender utilizes the personalized model to make recommendations on the current editing code.

5.4.2 Code history extractor

Because our recommendation techniques are learned from the personalized code history of programmers, we have built a code history extractor module for extracting usages of variables, methods, classes, and parameters of a programmer from his code development history. Typically, whenever a programmer adds a new code or updates the current existing code, he will submit a commit to the version control system. Fig 5.3 shows an example of code changes in a commit of a programmer. In the example, the programmer switched from using a HTTPResponse object to a HttpUrlConnection object. In our approach, we extract personalized code patterns from code changes in commits. In particular, for each code change

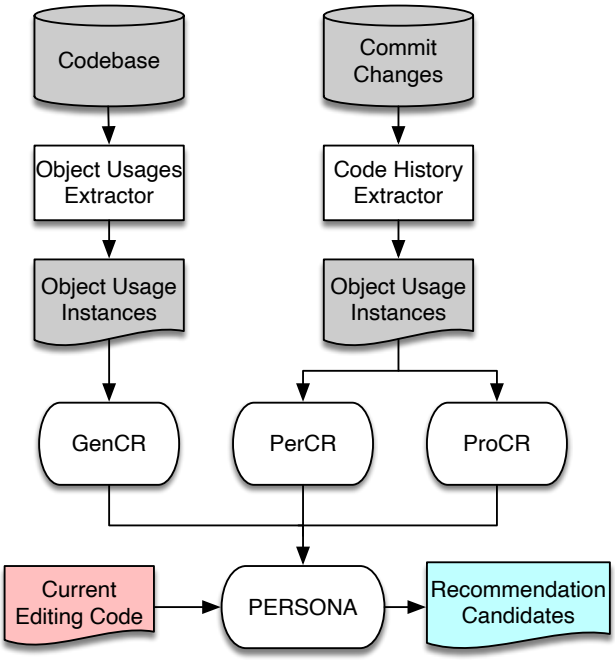

Figure 5.2: Overview of the system

in a commit of the programmer, the code history extractor will analyze the post-commit version, and extract object usages in the new code that the programmer added.

COMMIT MESSAGE: “Switched to HttpURLConnection”

```
try {
-    HttpResponse httpResponse = getHttpClient().execute(httpGet);
-    InputStream responseInputStream = httpResponse.getEntity().getContent();
-    return new WebResourceResponse(
-                        httpResponse.getEntity().getContentType().toString(), "UTF-8",
-                        responseInputStream);
+    URL imageUrl = new URL(url);
+    HttpURLConnection conn = (HttpURLConnection) imageUrl.openConnection();
+    conn.setReadTimeout(WPRestClient.REST_TIMEOUT_MS);
+    conn.setConnectTimeout(WPRestClient.REST_TIMEOUT_MS);
+    conn.setRequestMethod("GET");
+    conn.setRequestProperty("Authorization", "Bearer " + mToken);
+    conn.setRequestProperty("User-Agent", WordPress.getUserAgent());
+    conn.connect();
+    return new WebResourceResponse(conn.getContentType(), "UTF-8",
+                                   conn.getInputStream());
} catch (IOException e) {
-    AppLog.e(AppLog.T.READER, "Invalid reader detail request: " + e.getMessage());
+    AppLog.e(AppLog.T.READER, e);
}
```

Figure 5.3: An example of code changes in a commit

To extract the usages of variables, methods, classes, and parameters, the extractor uses GROUM (Graph-based Object Usage Model) [77] to represent the object usages in the source code. GROUM is a graph that represents the object usages in source code. It has two kinds

of nodes: *object nodes* and *action nodes*. An object node represents an object. It is labeled by the name of the object type (e.g. HttpURLConnection). An action node represents a method call. It is labeled the method qualified name (e.g. URL.openConnection). There are two kinds of edges representing control flow between action nodes and data flow between action nodes and object nodes.

In GROUM, each object created or involved during the execution is represented as an object node. We also treat primitive variables as object nodes. Action nodes represent any action that is performed on object nodes. Action nodes could be object instantiations, method calls, data field accesses of one object, or other operations. Object nodes are labeled by class names (object nodes represent primitive variables are labeled by types). Action nodes of types object instantiations, method calls, or data field accesses are labeled as "C.m" where C is its class name and m is the method (or field) name. Other action nodes that represent operations are labeled as the name of the operation.

The control edges of GROUM are used to represent the temporal orders between action nodes. A control edge from an action node A to action node B means that A is executed before B in the execution path. Because GROUM is defined for each execution path, thus, there is only one temporal order between action nodes, which is represented by a set of control edges between action nodes. The data edges indicate the data dependencies between data nodes and action nodes. A data edge from object node A to action node B means that A is a parameter of the action that B represents. A data edge from an action node B to data node A means the action B returns the object node A.

Fig 5.4 illustrates the GROUM of the code that the programmer added in the commit. Rectangle nodes are action nodes, while object nodes are represented as round rectangle nodes. Solid arrows represent the control edge between action nodes and dashed arrows represent data edges. The algorithms are used to extract GROUM from source code could be found at [83, 81, 77].

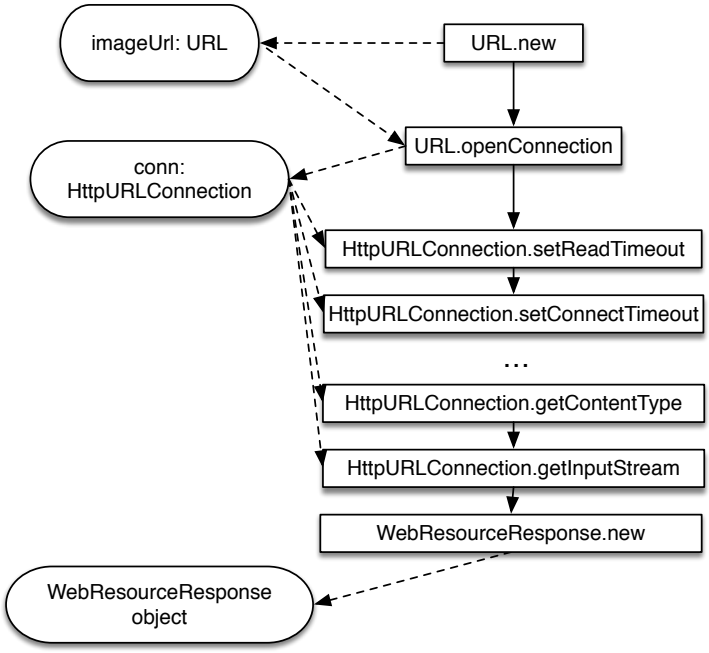

Figure 5.4: The extracted GROUM of the added code

There are several advantages of using GROUM to represent and extract the usages of variables, methods, classes, and parameters. Firstly, it removes redundant information in code such as keywords (try, return,...), or symbols (=,+...), and only focuses on important information such as objects, method calls. Secondly, GROUM avoids the problem of duplicated counting when extracting the occurrences of code elements and co-occurrences between code elements. For example, in Fig 5.4, the variable name conn appears multiple times in the code. Using GROUM, all the appearances are traced back to a single object node. Thus, the occurrence of conn is counted as one and the co-occurrence between conn and other code elements also counted as one.

To produce the training data for Persona, the extractor travels through nodes in GROUM and counts the occurrences and co-occurrences between code elements. Note that, for each occurrence of a code element, we also store the timestamp in which the programmer added it to the project. The time information is important when training the personalized model. To train the sub-model GenCR, we also developed a code extractor to extract GROUM from source files in a code corpus.

5.4.3 Learning recommendation models

We train each sub-model in PERSONA separately. To train the personalized model PERCR, we need to calculate $n_{f,c}(H_d), n_f(H_d), n_c(H_d)$ in Equation 5.2. Calculating these values requires counting the occurrences of code elements and co-occurrences between code elements in the code history of the programmer. We explained the counting process in the previous section. Training the project-level model PROCR is similar to PERCR, the only difference is that PROCR is trained on the code history of other programmers in the current project. To train GENCR, we collect a code dataset contains multiple projects. Next, we obtain the source files from the projects and extract GROUM from source files. GENCR is trained by computing values $n_{f,c}(\overline{P}), n_f(\overline{P}), n_c(\overline{P})$ described in Equation 5.4. Finally, we incorporate the three sub-models with either Equation 5.5 or 5.6.

5.4.4 Recommending code

Let us go back to the scenario in Fig 5.1. The programmer writes code to read a file. In the first line, he creates an InputStream object from the filename. Next, he creates an InputStreamReader object from the InputStream. Let us assume, he invokes code recommendation at the first of line 4. Upon the request, our tool will analyze the current edding code, build a temporary GROUM, and extract the set of context features F includes object types {InputStream, InputStreamReader}, variables {is, testContext, inputStreamReader}, method calls {getAssets, open, InputStreamReader.new }, and parameters {fileName}.

In the next step, the tool will build a set of candidates for recommendation. It starts by analyzing which types of code elements are asked for the recommendation. In the example, the candidates should be a class or a variable. All classes and variables that are available in the current editing code will be added to the set of candidates. The tool then utilizes PERSONA to calculate the relevance score of each recommendation candidate towards the set of context features F using Equation 5.8. The set of candidates will be sorted by the relevance score. Finally, the recommendation tool returns the rank list of candidates with relevant

scores for the programmer to consider. Note that, if the user requests recommendations for a new variable name, the tool will consider all the variable names has been used for the object before as the candidates. These names are stored in our model and might not be in the editing code.

5.5 Evaluation

We conducted several experiments to evaluate the effectiveness of our approach to learning and recommending code for programmers. All experiments are executed on a computer running Windows 10 with Intel Core i7 3.6Ghz CPU, 16GB RAM, and 1TB HDD storage. To conduct the evaluation, we collected a dataset consists of multiple Java projects that have source code repositories available on GitHub. The dataset that we used was carefully collected and studied by Allamanis *et al.* [5]. The corpus can be found and downloaded online [4]. It contains 14,807 projects across a wide variety of domains amounting to over 350 million lines of code in over 2 million files. The number of code tokens in the dataset exceeds 1.5 billion. Note that the dataset only contains the .java extension files, it does not contains revisions or commit changes. The characteristics of the corpus are shown in Table 5.1. We call this dataset A14K.

Table 5.1: Dataset characteristics

Number of projects	14,807
Number of files	2,130,264
Number of lines of code	352,312,696
Number of code tokens	1,501,614,836

Because the A14K dataset only contains a snapshot of .java files, it is only used for training GenCR and the baselines. For evaluating Persona, it is required to have the code history of projects and programmers. Thus, we manually selected 10 projects in the dataset to evaluate our model. We selected such projects by first sorting the projects in the dataset by the number of commits. Next, we chose projects that have the highest number of commits

while the vast majority of code is written in Java. We avoided selecting certain projects. First, we avoided projects that share duplicated code with a previously selected project. We also avoided projects are developed in multiple programming languages. For each selected project, we checked out its source code repository to retrieve all the code and commit changes. Table 5.2 shows the list of selected projects along with the number of contributors and commits.

Table 5.2: Projects used in the evaluation

Name	Description	#Contributors	#Commits
liferay-portal	Enterprise web platform	553	407,015
intellij-comm	IDE for Java	510	279,093
osmand	Android map app	688	62,493
geogebra	Multi-platform math app	24	52,750
elasticsearch	Search engine	1,415	52,628
camel	Middleware framework	609	45,158
lucene-solr	Enterprise-search platform	181	33,654
cdt	IDE for C++	266	34,603
cassandra	Database system	298	25,289
hadoop	Data processing framework	278	23,877

5.5.1 Settings and baselines

For each selected project, the set of commits is sorted in chronological order. Next, we grouped commits by programmers. When we perform an evaluation experiment for a programmer d, his commit set is divided into a training set TR_d and testing set TE_d chronologically. The training set TR_d is used to train the sub-model PerCR. The code in the testing set TE_d is used to evaluate later. In our experiments, the training set is selected as the first 70% of the initial set of commits in the chronological order, while the remaining 30% of the commits are used as the testing set. The sub-model ProCR is trained on the set of commits of other programmers. The commits that are used to train PerCR appear before the first commit of the testing set TE_d. The sub-model GenCR is trained on the initial dataset, which contains a snapshot of .java files (the current project is excluded). Finally, we combine the sub-models using both Equation 5.5 and 5.6. In the first method, we set the

weighting coefficients equally, $\alpha_1 = \alpha_2 = \alpha_3 = \frac{1}{3}$. The second method uses the max function to combine sub-models. The two approaches are called PersonaSum, and PersonaMax correspondingly.

In our evaluation, to compare our model with the baselines, we chose the task of recommending the next identifier in a code sequence. The types of identifiers that we considered include variable and field names, type names such as class and interface names, method names, and parameters. Given a code sequence, a recommendation model is expected to recommend the most probable identifier. Alamalis *et al.* [5] shows that learning to predict code elements is difficult mainly because of the identifiers. Thus, we chose this task to better compare the effectiveness of recommendation models. This evaluation task has been used similarly in the evaluation of prior approaches [39, 104, 76].

Recommendation accuracy is measured as follows. Our evaluation tool predicts and evaluates *all identifiers* in every code sequence from the testing set. At a position i, it uses the recommendation model under evaluation to compute the top k most likely identifiers $x_1, x_2, ..., x_k$ for that position based on the previous code tokens. If the actual identifier s_i at position i is among k suggested results, we count this as a hit. The top-k suggestion accuracy for a sequence is the ratio of the total hits over the sequence's length. For example, if we have 70 hits on a code sequence of lengths 100 for a test file, accuracy is 70%. The top-k accuracy is the ratio of the total hits over the total number of evaluated tokens.

To compare the effectiveness of Persona, we chose two baseline models: n-gram and recurrent neural network (RNN) for comparison due to the following reasons. First, both of them are popular statistical models to capture common patterns in a large dataset and are comparable with Persona. In addition, n-gram is widely used in recent research on code recommendation [39, 104, 76]. Raychev *et al.* [92] and White *et al.* [111] recently evaluated RNN and n-gram in code recommendation and reported RNN as the better approach. Note that our model and the baselines use the same 14K dataset as the training set.

An n**-gram model** is a simple statistical model for modeling sequences. An n-gram model learns all possible conditional probabilities $P(m_i|m_{i-n+1}...m_{i-1})$, where m_i is the current code token and $m_{i-n+1}...m_{i-1}$ is the sub-sequence of $n-1$ prior tokens. This is the probability that m_i occurs as the next code tokens of $m_{i-n+1}...m_{i-1}$. Using the chaining rule, we can use an n-gram model to compute the generating probability of any given sequence $m_1...m_n$. To improvement the effectiveness of n-gram model, Tu *et al.* [104] introduced CacheLM, a novel cache language model that consists of both an n-gram and an added "cache" component to exploit localness. The cache is the set of code tokens that appear in the same project with the test file. We re-implement this method for comparison. We used the same settings with the original model, i.e. 3-gram with "5K tokens" cache size. The model is trained on the A14K dataset with the cache is extracted from the project under test.

A **Recurrent neural network** (RNN) is a class of neural networks for learning sequences. A single-layer RNN can be trained with a collection of code token sequences and then is able to compute the probability of the next code token for any given sequence. In other words, the RNN can compute all conditional probabilities $P(m_i|m_1...m_{i-1})$ for any given sequence $m_1...m_n$. To do that, it maintains a context vector (hidden state) c_i represents current context of sub-sequence up to $m_1...m_{i-1}$. A function f is learned from data to compute the context vector at position i, $c_i = f(m_i, c_{i-1})$ given the current token m_i and previous context c_{i-1} while another function g is learned to compute the probability of the next token m_{i+1}, $P(m_{i+1}|m_1...m_i) = g(c_i)$ given the current context c_i. To improve the modeling performance, we could stack multiple layers of RNNs on top of each other to create a Deep RNN. Each hidden state is continuously passed to both the next time step of the current layer and the current time step of the next layer. The model could still be further improved by using a special type of the hidden layer called Long Short-Term Memory (LSTM) cell to tackle the problem of unstable gradients and handle long sequences. A Deep RNN model with too many hidden layers is quite computationally expensive. Thus, in our experiment, we implemented a model with a stack of 5 hidden states, each hidden state is

an LSTM cell with 200 hidden units. We call this model DRnn200-5. We implemented DRnn200-5 using Keras Sequential APIs, TensorFlow 2, and running a Google Colab Pro machine. Note that, the code sequences are used to train both CacheLM and DRnn200-5 are extracted using the GROUM model as we described in Section 5.

5.5.2 Recommendation accuracy

In this section, we show the recommendation accuracy of our proposed models and the baselines on top-contributed programmers over 10 selected projects. For each project, we select top-5 contributed programmers by the number of commits. We train and test our models and the baselines for each programmer. Due to a lack of space, we only report top-1 accuracy. Tables 5.3 and 5.4 show the top-1 commendation accuracy of top-contributed programmers in the projects intellij-comm and osmand. To report the result in all the selected projects, we compute the average top-1 accuracy of programmers in each project. Table 5.5 shows the average results in all 10 projects.

Table 5.3: Recommendation accuracy of top-contributed programmers in intellij-comm

Name	#Commits	PerCR	ProCR	GenCR	PersonaSum	PersonaMax	CacheLM	DRnn200-5
D. Jemerov	15,962	58.5%	55.7%	41.7%	64.3%	63.6%	47.0%	57.5%
Peter	14,615	55.9%	53.2%	39.2%	66.0%	62.9%	48.0%	58.3%
A. Kudravtsev	13,934	59.3%	55.3%	40.1%	64.9%	63.0%	47.1%	57.9%
A. Kozlova	12,589	57.5%	55.8%	39.8%	66.0%	61.4%	46.3%	58.4%
V. Krivosheev	11,784	55.7%	57.0%	40.8%	66.6%	62.5%	47.0%	58.8%

Table 5.4: Recommendation accuracy of top-contributed programmers in osmand

Name	#Commits	PerCR	ProCR	GenCR	PersonaSum	PersonaMax	CacheLM	DRnn200-5
Victor Shcherb	6,730	53.8%	49.6%	39.1%	63.1%	61.4%	48.2%	55.4%
Weblate	4,029	50.4%	48.8%	38.3%	61.8%	58.4%	47.0%	55.3%
Sonora	3,586	51.9%	47.8%	37.9%	60.5%	60.9%	47.2%	54.4%
Franco	2,536	52.8%	49.2%	34.3%	59.4%	60.8%	45.3%	53.2%
Jan Madsen	2,207	50.1%	46.8%	38.5%	61.7%	62.6%	46.4%	56.6%

From the tables, we can see several interesting results. Overall, the personalized model PerCR outperforms the project-level recommendation model ProCR, and the general model GenCR. It generates a 2-3% gap over ProCR and 12-15% gap over GenCR. When

Table 5.5: Recommendation accuracy of top-contributed programmers in selected projects

Project	PerCR	ProCR	GenCR	PersonaSum	PersonaMax	CacheLM	DRnn200-5
liferay-portal	53.7%	51.5%	37.5%	65.3%	63.4%	46.9%	57.9%
intellij-comm	57.4%	55.4%	40.3%	65.6%	62.7%	47.1%	58.7%
osmand	51.8%	48.0%	37.6%	61.3%	60.8%	46.8%	54.3%
geogebra	52.6%	48.5%	35.5%	61.7%	59.8%	46.7%	54.4%
elasticsearch	46.8%	43.8%	36.9%	58.4%	56.4%	42.0%	54.6%
camel	49.5%	45.6%	38.8%	58.1%	56.8%	43.3%	53.2%
lucene-solr	50.2%	49.1%	40.9%	60.5%	58.3%	42.5%	54.9%
cdt	48.3%	44.9%	37.9%	57.2%	55.5%	44.1%	51.8%
cassandra	49.5%	45.5%	37.6%	56.1%	55.2%	43.5%	52.4%
hadoop	50.5%	46.7%	40.9%	60.6%	58.5%	45.7%	54.7%

incorporating sub-models together, the recommendation accuracy increases significantly with top-1 accuracy approaching 60-65%. The combining method using weighting coefficients (PersonaSum) yields a slightly higher result compared to using the max function (PersonaMax) but the difference is insignificant.

About the baselines, the top-1 accuracy of DRnn200-5 is significantly higher than CacheLM with a gap of around 10%. This shows that DRnn200-5 is a much better approach for modeling sequences. The top-1 accuracy of DRnn200-5 is also better when compared with each individual sub-model. However, when combining the sub-models together, PersonaSum still has higher top-1 accuracy than DRnn200-5 by an average of 4-6%. Overall, the result shows that by combining three simple sub-models that capture personal, project-specific, and common code patterns together, Persona still outperforms the baselines which mostly focus on the common code patterns.

5.5.3 Recommendation accuracy over time

In this section, we evaluate the recommendation accuracy of Persona and sub-models overtime. We design the experiment as follows. For a programmer d, we divide his set of commits into equal time intervals. A time interval of t_i contains all the commits of the programmer during that time. Depending on the code history of programmers, we could divide the commits into months, quarters, or years. At time interval t_i, we will use all the commits of the programmer before t_i to train PerCR, and all the commit of other

programmers before t_i to train ProCR. In other words, Persona is trained on all the commits of the project before t_i. All the commits in t_i will be used for testing. With this experiment, we want to replicate the real-world accuracy of code recommendation models.

We choose intellij-comm as the subject system. The project has 279,093 commits with 510 contributors spanning from 2005 to now. From the project, we select three programmers with different types of contributions for evaluation. Due to the lack of presentation space, we will use quarters as the time interval, and we only show the recommendation accuracy for the first 20 intervals. Figs 5-7 show the top-1 recommendation accuracy overtime for three programmers in the project. For better visualization, we only shows the top-1 accuracy of GenCR, ProCR, and Persona in the figures. The top-1 accuracy of GenCR and the baselines is similar to values reported in Table 5.3.

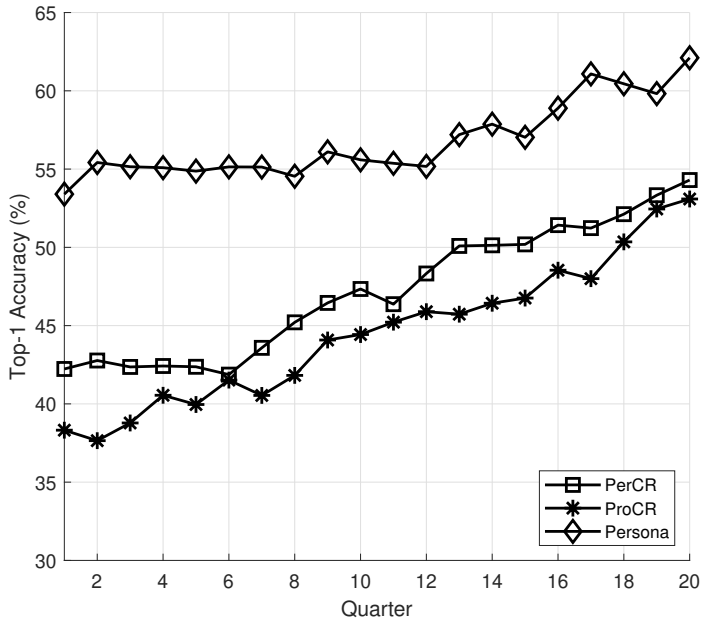

Figure 5.5: Top-1 accuracy over time of "Dmitry Jemerov"

From the figures, we can see that the recommendation accuracy of PerCR and ProCR increase over time as these models have more training data. This leads to an increase in the accuracy of Persona. Another interesting observation is that the amount of training data affects PerCR and ProCR significantly. The first programmer (Fig 5.5) is the main

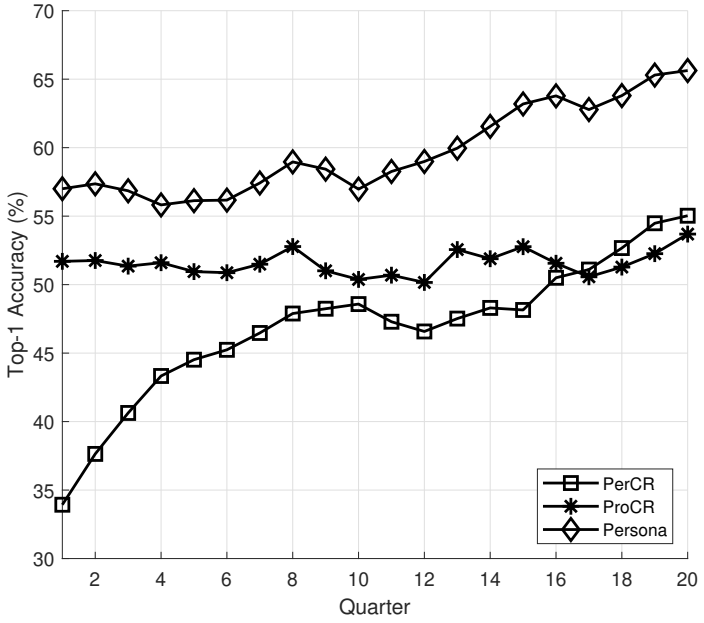

Figure 5.6: Top-1 accuracy over time of "Vladimir Krivosheev"

contributor to the project from the start. As he committed a lot of code, his personalized model outperforms the project-level model. The second programmer (Fig 5.6) joined the project when it already contains most of the code. Thus, his project-level recommendation mode outperforms the personalized model in the beginning. Finally, the third programmer (Fig 5.7) has a limited contribution at the beginning of the project. His personalized model has low accuracy at the beginning due to a lack of training data. Overall, the experiment shows that the recommendation accuracy of Persona improves overtime as more training data is available.

5.5.4 Accuracy on lower-contributed programmers

We have studied the recommendation accuracy of our models on top-contributed programmers. In this section, we study how our models perform when recommending code for lower-contributed programmers. Of course, we do not want to select programmers that committed too little code, as we want to ensure that we have enough training and testing data for the personalized model. Thus, we select the programmers to evaluate as follows.

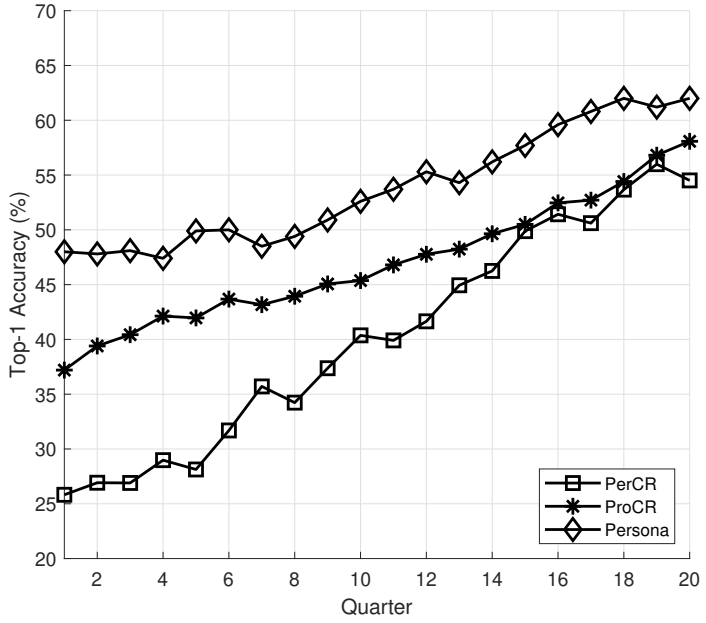

Figure 5.7: Top-1 accuracy over time of "Alexey Kudravtsev"

First, for each project, we filter out all programmers with less than 20 commits. Next, we sort programmers by the number of commits and find the median of the list. We select five programmers that have the number of commits right above the median for the study. We train and test our models and the baselines for each programmer. To report the result in all the projects, we compute the average top-1 accuracy of the five programmers in each project.

Table 5.6 shows the average results in all 10 projects. We could see that the top-1 accuracy of the personalized model PERCR is low due to the lack of training data. The project-level model PROCR still performs reasonably well when compared to other models. This could be explained that the lower-contributed programmers often join the project later when the project has been developed extensively, and they might reuse the project-specific code. Thus, the accuracy of the recommendation model is maintained. On average, PERSONA still achieves the highest top-1 accuracy when compared to the baselines. For example, PERSONASUM has higher top-1 accuracy than DRNN200-5 in 8 out of 10 selected projects with a gap of around 2-5%, while DRNN200-5 just slightly outperforms in the remaining 2 projects. Overall, by incorporating three sub-models together, the PERSONA performs

reasonably well even if the programmers have low contribution in the project or did not join the the project for a long time.

Table 5.6: Recommendation accuracy of lower-contributed programmers in selected projects

Project	PerCR	ProCR	GenCR	PersonaSum	PersonaMax	CacheLM	DRnn200-5
liferay-portal	26.5%	48.7%	37.5%	58.6%	56.8%	46.2%	51.3%
intellij-comm	24.2%	51.4%	39.8%	57.3%	57.1%	49.8%	54.4%
osmand	22.7%	45.3%	35.7%	57.0%	52.8%	47.2%	53.4%
geogebra	20.7%	44.8%	35.5%	56.3%	54.1%	46.0%	53.7%
elasticsearch	15.3%	41.4%	36.9%	47.2%	49.7%	41.1%	47.8%
camel	20.1%	43.6%	38.8%	52.4%	47.9%	42.9%	49.6%
lucene-solr	17.5%	46.5%	40.9%	53.9%	52.5%	42.0%	51.3%
cdt	21.7%	42.0%	37.9%	48.3%	48.7%	43.5%	49.2%
cassandra	15.6%	42.9%	37.6%	51.0%	49.6%	42.6%	49.3%
hadoop	16.5%	43.1%	40.9%	55.6%	52.4%	44.9%	51.4%

5.5.5 Ablation study

In this section, we perform an ablation study to understand the contribution of the sub-models to the recommendation performance of Persona. In particular, we focus on the model PersonaSum. Similar to the first experiment, we measure the recommendation accuracy of PersonaSum with different configurations on top-contributed programmers over 10 selected projects. For each project, we select top-5 contributed programmers by the number of commits, then we compute the average top-1 accuracy of programmers in each project. As described in Section 4, in PersonaSum, we combine sub-models using Equation 5.5 where $\alpha_1 + \alpha_2 + \alpha_3 = 1$ are weighting coefficients. The values of $\alpha_1, \alpha_2, \alpha_3$ represent the contribution level of PerCR, ProCR, and GenCR correspondingly. Removing a sub-model from the system equals to setting $\alpha_i = 0$. For example, if we remove ProCR from the system, α_2 is set to 0, which means $\alpha_1 = \alpha_3 = \frac{1}{2}$. The model is called PerCR+GenCR. Similarly, if we remove ProCR and GenCR from the system, α_2 and α_3 are set to 0, which means $\alpha_2 = 1$. The model become PerCR.

Table 5.7 shows the average top-1 recommendation accuracy when removing one or two sub-models from the system. Note that, as we use the same settings as the previous experiment, the results for PerCR, ProCR, GenCR, PersonaSum are the same as in

Table 5.5. We have several interesting observations. Firstly, if we remove two sub-models from the system, the sub-model PerCR outperforms the project-level recommendation model ProCR, and the general model GenCR. Secondly, if we remove a sub-model from the system, PerCR+GenCR has the highest top-1 accuracy. PerCR+GenCR also has a significantly higher top-1 accuracy when compared to then each individual sub-model, especially, GenCR. This result shows that although GenCR has low top-1 accuracy, combining this sub-model with others could improve the recommendation significantly. Finally, we could see that PerCR+ProCR does not have much improvement when compared to each sub-model.

Table 5.7: Recommendation accuracy by removing one or two sub-models from Persona

Project	PerCR	ProCR	GenCR	PerCR+ProCR	ProCR+GenCR	PerCR+GenCR	PersonaSum
liferay-portal	53.7%	51.5%	37.5%	56.6%	55.8%	61.1%	65.3%
intellij-comm	57.4%	55.4%	40.3%	59.1%	58.3%	60.7%	65.6%
osmand	51.8%	48.0%	37.6%	54.3%	50.2%	55.9%	61.3%
geogebra	52.6%	48.5%	35.5%	53.9%	52.1%	56.8%	61.7%
elasticsearch	46.8%	43.8%	36.9%	48.9%	47.3%	53.8%	58.4%
camel	49.5%	45.6%	38.8%	51.1%	48.5%	53.4%	58.1%
lucene-solr	50.2%	49.1%	40.9%	52.6%	51.9%	57.4%	60.5%
cdt	48.3%	44.9%	37.9%	50.6%	48.6%	54.0%	57.2%
cassandra	49.5%	45.5%	37.6%	49.9%	49.2%	53.8%	56.1%
hadoop	50.5%	46.7%	40.9%	51.5%	50.8%	56.1%	60.6%

5.5.6 Weighting coefficients

In this section, we study how the values of weighting coefficients affect the recommendation result of the model. As a personalized model, Persona is built and updated for each programmer. We selected the top-1 programmer by the number of commits in the intellij-comm project to study the weighting coefficients. Let us assume we choose the weighting coefficient α_1 to study. Note that the weighting coefficients have a constraint $\alpha_1 + \alpha_2 + \alpha_3 = 1$. For each value of α_1, we set $\alpha_2 = \alpha_3 = \frac{1-\alpha_1}{2}$. Next, we let α_1 takes different values from 0 to 1, and increase by 0.1. Then we evaluate the top-1 accuracy of PersonaSum based at each value of α_1. Fig 5.8 shows how top-1 accuracy changes when α_1 changes. A similar process is repeated for the remaining weighting coefficients. Figs 5.9 and 5.10 show the result for α_2 and α_3.

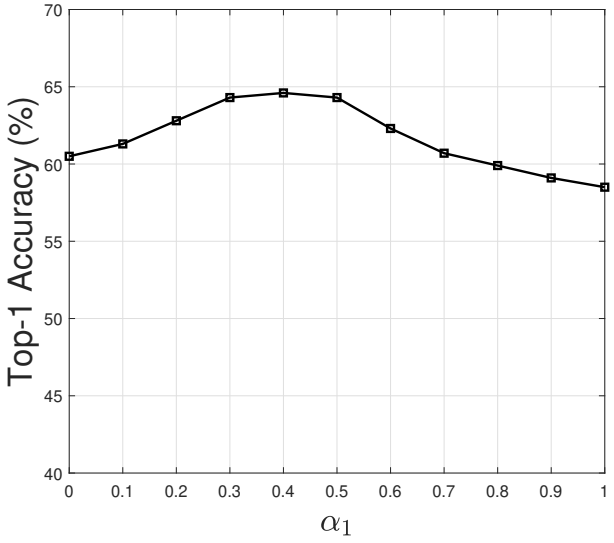

Figure 5.8: Top-1 accuracy when α_1 changes

From the figures, we have several observations. Firstly, if a weighting coefficient has a high value (closer to 1), the top-1 accuracy tends to decrease. In such a case, the top-1 accuracy of PersonaSum is dominant by a sub-model. Secondly, the top-1 accuracy result is most sensitive with the value of α_3. When α_3 is high, the result decreases significantly as the weighting of GenCR increases in PersonaSum. Additionally, we could see that the result is often high if all the weighting coefficients are in the range $[0.3, 0.5]$. These observations are valuable in selecting the values of weighting coefficients to improve the recommendation accuracy of Persona.

5.6 Discussion

In this section, we discuss several aspects of Persona in more detail. From the machine learning perspective, Persona is a simple ensemble approach with three sub-models: PerCR, a model that captures personal code patterns of a programmer; ProCR, a model that captures the project-level code patterns that the programmer is working on; and GenCR, a general model that capture code patterns shared between multiple projects. Persona learns and recommends like n-gram and RNN but more flexibly. For example,

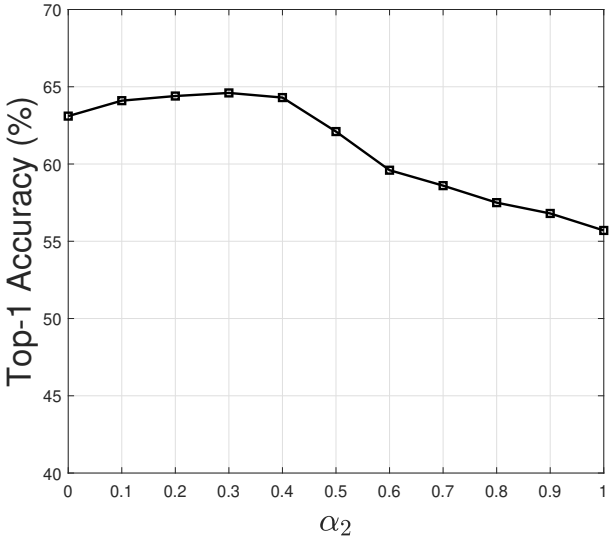

Figure 5.9: Top-1 accuracy when α_2 changes

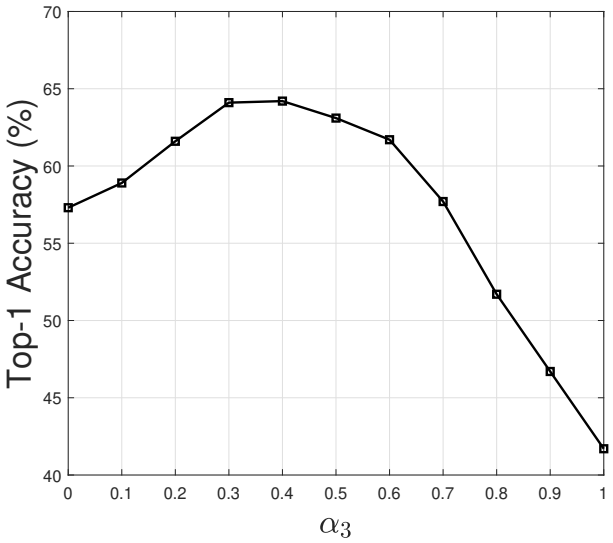

Figure 5.10: Top-1 accuracy when α_3 changes

fuzzy membership functions are not probability distribution functions, thus, PERSONA does not need to normalize $\sum_e \mu_f(c) = 1$.

We could consider PERSONA is a fuzzy logic system specially designed for the software engineering domain. It represents code patterns as fuzzy logic rules. It uses fuzzy set theory to model and apply those rules and uses fuzzy union operations to combine the rules. In the traditional fuzzy logic system, variables are often continuous such as Temperature, Density or

linguistic like LOW, VERY LOW. The membership functions are often manually defined by domain experts with functions such as triangular or trapezoidal. In PERSONA, variables are discrete, i.e. class, method, etc., and the membership functions are estimated automatically.

In our evaluation, we re-implemented (CACHELM) as the baseline method. Although we tried to replicate the same settings as the previous research [92, 104], the recommendation results of the baseline models in our evaluation are different when compared to the original research. The dissimilarity could be explained due to the differences in several factors including the dataset, cross-validation, recommendation tasks, etc. Similarly, in our implementation of DRNN200-5, we used different configurations with the previous studies [92, 111] so the result is not comparable.

The evaluation suggests that PERSONA outperforms the baselines such as DRNN200-5 which reaffirms our earlier assumption. As a crowd-based approach, DRNN200-5 infers and recommends common code patterns from a large code corpus while ignoring the difference in coding preferences of programmers. When such differences are blurred, the performance of the recommendation tool for a specific programmer is hurt. PERSONA achieves high accuracy because it takes into consideration the personal coding preferences of programmers while also captures the project-specific and common code patterns. In our future work, we plan to incorporate personal coding patterns with the model such as DRNN200-5 to further improve the recommendation accuracy.

In PERSONA, we combine the sub-models using both Equations 5.5 and 5.6. In the first method, we set the weighting coefficients equally, $\alpha_1 = \alpha_2 = \alpha_3 = \frac{1}{3}$. These weighting coefficients determine the contribution of a sub-model to PERSONA. We performed a study on how values of weighting coefficients affect the recommendation result of the model, which reveals several insights. Different combinations of weighting coefficients could be experienced to optimize the performance of the model. In the future work, we plan to develop a method to estimate such optimal coefficients.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Software systems and services play a vital part in the work and life of billions of people around the world. With the huge demand, modern software development often relies heavily on API frameworks and libraries such as Android and iOS frameworks, Java APIs, etc. and new applications extensively re-use API components. However, learning to use API objects and methods is usually challenging for developers due to a large number of API methods and objects, the constant changing and complexity of APIs, and the incompleteness of API documentation. In this dissertation, we develop three statistical models that capture API usage patterns. The approaches were trained on thousands of code projects and help to improve programming tasks including code recommendation, code verification, exception handling, bug detection, and bug fixing.

One is Salad, a novel approach to learn API usages from bytecode of Android mobile apps. the core contributions of the approach include HAPI, a statistical model of API usages and three algorithms to extract method call sequences from apps' bytecode, to train HAPI based on those sequences, and to recommend method calls in code completion using the trained HAPIs. Our empirical evaluation shows that our code recommendation tool based on Salad can effectively learn API usages from 200 thousand apps containing 350 million method sequences. It recommends next method calls with top-3 accuracy of 90% and outperforms baseline approaches on average 10-20%.

Secondly, we introduce FuzzyCatch, a code recommendation tool for handling exceptions. Based on fuzzy logic, FuzzyCatch can predict if a runtime exception would occur in a given code snippet and recommend code to handle that exception. FuzzyCatch

is implemented as a plugin for Android Studio. The empirical evaluation suggests that FuzzyCatch is highly effective. For example, it has top-1 accuracy of 77% on recommending what exception to catch in a try catch block and of 70% on recommending what method should be called when such an exception occurs. FuzzyCatch also achieves a high level of accuracy and outperforms baselines significantly on detecting and fixing real exception bugs.

Finally, we propose Persona, a personalized code recommendation model. It learns personalized code patterns for each programmer based on their coding history, while also combines with project-specific and common code patterns. Persona supports recommending code elements including variable names, class names, methods, and parameters. The empirical evaluation suggests that our recommendation tool based on Persona is highly effective. It recommends the next identifier with top-1 accuracy of 60-65% and outperforms the baseline approaches.

6.2 Future work

In this section, we depict some extension of our current work as follows:

Finding and fixing API misuses. Application Programming Interface (API) libraries and frameworks play an important role in modern software development. There are certain rules and constraints when using API objects and methods in such libraries. Violating such constraints (i.e., API misuse) could cause serious programming errors such as system crashing or resource leaking. To solve the problem, researchers have proposed and developed several API misuse detectors [7, 10]. Unfortunately, recent studies show that those detectors often still have low accuracy [8]. In addition, to the best of our knowledge, no current approach could repair the detected API misuses automatically. To understand the nature of API misuses, we performed a preliminary study [86] on a dataset of API misuses provided by Amann *et al.* [7]. We found that the root causes of those API misuses can be classified into five groups including incorrect temporal order of method calls, incorrect handling of exceptions, missing pre-conditions and post-conditions, and incorrect values of arguments.

In the future work, we plan to propose an approach that uses several statistical models for five factors involving the usage of any API method call: the temporal order between method calls, exception handling, pre-condition, post-condition, and argument values. Those statistical models are trained using a massive amount of high-quality production code available in open-source repositories and app stores. After training, the approach can detect and repair API misuses in a given code snippet. It first validates every method call in such code using its trained statistical models. If one usage factor of a method call m has a sufficiently low probability (e.g. less than a threshold), it considers the call as an API misuse. The API misuse correction problem could be seen as an optimal search problem in which the approach searches for repair actions that optimally eliminate those low probability usage factors.

User interface design patterns. User interface (UI) is one of the most important components of an app because users mainly interact with the app via UI (e.g., tap/click on an icon or scroll/swipe a page). Thus, UI strongly influences users' perception of and experience with the app, which could decide whether the app is successful or not. To design beautiful, appealing apps, app development teams often hire well-trained UI/UX designers. However, even with high-skilled UI designers, the UI design tasks are mostly manual, and thus, very time-consuming. To address this problem, we plan to propose an approach to (semi)-automate those tasks based on learning UI design patterns. A UI design pattern is a design idea or template that is used in different apps (or in different places in the same app). For example, most apps have a login screen with UI elements for user id (e.g., username, email, or phone number), password, and sometimes third-party authentication methods (e.g., login via Facebook, Google, or Twitter accounts). The key idea of the approach is to develop and deploy advanced deep learning models based on recurrent neural network (RNN) and generative adversarial network (GAN) to learn UI design patterns from millions of mobile apps currently available on app stores. After learning, those models can be used to support app developers in designing UI of new apps with the UI design patterns they learned.

Bibliography

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] J. M. Al-Kofahi, A. Tamrawi, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Fuzzy set approach for automatic tagging in evolving software. In *ICSM*, 2010.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the ACM SigSoft Symposium on Foundations of Software Engineering*. ACM – Association for Computing Machinery, November 2014.

[4] M. Allamanis and C. Sutton. Github java corpus.

[5] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 207–216, May 2013.

[6] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM.

[7] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. Mubench: A benchmark for api-misuse detectors. In *MSR*, Austin, USA, 2016.

[8] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *TSE*, 2018.

[9] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. Investigating next steps in static api-misuse detection. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 265–275. IEEE Press, 2019.

[10] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. Investigating next steps in static api-misuse detection. In *MSR*, Montreal, Canada, 2019.

[11] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. CSCC: simple, efficient, context sensitive code completion. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 71–80, 2014.

[12] E. A. Barbosa and A. Garcia. Global-aware recommendations for repairing violations in exception handling. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 858, New York, NY, USA, 2018. Association for Computing Machinery.

[13] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa. Categorizing faults in exception handling: A study of open source projects. In *2014 Brazilian Symposium on Software Engineering*, pages 11–20, Sep. 2014.

[14] E. A. Barbosa, A. Garcia, and M. Mezini. Heuristic strategies for recommendation of exception handling code. In *Proceedings of the 2012 26th Brazilian Symposium on Software Engineering*, SBES '12, page 171–180, USA, 2012. IEEE Computer Society.

[15] E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus. Enforcing exception handling policies with a domain-specific language. *IEEE Transactions on Software Engineering*, 42(6):559–584, 2016.

[16] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[17] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 242–251, New York, NY, USA, 2006. ACM.

[18] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

[19] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. Ejflow: Taming exceptional control flows in aspect-oriented programming. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, AOSD '08, page 72–83, New York, NY, USA, 2008. Association for Computing Machinery.

[20] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora. A hidden markov model to detect coded information islands in free text. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 157–166, Sept 2013.

[21] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *Software Engineering, IEEE Transactions on*, 34(5):579–596, Sept 2008.

[22] I. Chawla and S. K. Singh. An automated approach for bug categorization using fuzzy logic. In *ISEC*, 2015.

[23] H. Chen, W. Dou, Y. Jiang, and F. Qin. Understanding exception-related bugs in large-scale cloud systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 339–351, Nov 2019.

[24] W. Chu and S.-T. Park. Personalized recommendation on dynamic content using predictive bilinear models. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, page 691–700, New York, NY, USA, 2009. Association for Computing Machinery.

[25] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *MSR*, 2015.

[26] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 134–145, Piscataway, NJ, USA, 2015. IEEE Press.

[27] G. B. d. Pádua and W. Shang. Revisiting exception handling practices with exception flow analysis. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 11–20, Sep. 2017.

[28] G. B. de Pádua and W. Shang. Studying the relationship between exception handling practices and post-release defects. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 564–575, New York, NY, USA, 2018. ACM.

[29] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[30] F. Ebert, F. Castor, and A. Serebrenik. An exploratory study on exception handling bugs in java programs. *J. Syst. Softw.*, 106(C):82–101, Aug. 2015.

[31] B. Eken. Assessing personalized software defect predictors. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 488–491, New York, NY, USA, 2018. Association for Computing Machinery.

[32] J. L. M. Filho, L. Rocha, R. Andrade, and R. Britto. Preventing erosion in exception handling design using static-architecture conformance checking. In *Software Architecture*, 2017.

[33] C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering (ICSE'07)*, pages 230–239, May 2007.

[34] Google. Android studio.

[35] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 332–343, Washington, DC, USA, 2009. IEEE Computer Society.

[36] A. E. Hassan, A. Hindle, P. Runeson, M. Shepperd, P. T. Devanbu, and S. Kim. Roundtable: What's next in software analytics. *IEEE Software*, 30(4):53–56, 2013.

[37] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer New York Inc., 2001.

[38] R. Hill and J. Rideout. Automatic method completion. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 228–235, Washington, DC, USA, 2004. IEEE Computer Society.

[39] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[40] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[41] D. Hou and D. M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *ICSM*, pages 233–242. IEEE, 2011.

[42] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 49–65, New York, NY, USA, 2014. ACM.

[43] http://developer.android.com.

[44] https://code.google.com/p/smali/.

[45] https://github.com/Akdeniz/google-play crawler.

[46] C.-S. Hwang, Y.-C. Su, and K.-C. Tseng. Using genetic algorithms for personalized recommendation. In J.-S. Pan, S.-M. Chen, and N. T. Nguyen, editors, *Computational Collective Intelligence. Technologies and Applications*, pages 104–112, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[47] F. Jacob and R. Tairas. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 104:1–104:6, New York, NY, USA, 2010. ACM.

[48] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289, 2013.

[49] M. Kechagia, M. Fragkoulis, P. Louridas, and D. Spinellis. The exception handling riddle: An empirical study on the android api. *Journal of Systems and Software*, 142, 04 2018.

[50] M. Kechagia, T. Sharma, and D. Spinellis. Towards a context dependent java exceptions hierarchy. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 347–349, Piscataway, NJ, USA, 2017. IEEE Press.

[51] M. Kechagia and D. Spinellis. Undocumented and unchecked: Exceptions that spell trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 312–315, New York, NY, USA, 2014. ACM.

[52] J. Kim, S. Lee, S. won Hwang, and S. Kim. Towards an intelligent code search engine. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.

[53] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE*, 2007.

[54] F. Kistner, M. Beth Kery, M. Puskas, S. Moore, and B. A. Myers. Moonstone: Support for understanding and writing exception handling code. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 63–71, Oct 2017.

[55] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[56] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan. Eh-recommender: Recommending exception handling strategies based on program context. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 104–114, Dec 2018.

[57] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.

[58] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.

[59] X. Liu, B. Shen, H. Zhong, and J. Zhu. Expsol: Recommending online threads for exception-related bug reports. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 25–32, Dec 2016.

[60] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 109–118, Washington, DC, USA, 2008. IEEE Computer Society.

[61] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *The 31st International Conference on Machine Learning (ICML)*, June 2014.

[62] C. Marinescu. Are the classes that use exceptions defect prone? In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, page 56–60, New York, NY, USA, 2011. Association for Computing Machinery.

[63] C. McMillan, D. Poshyvanyk, and M. Grechanik. Recommending source code examples via api call usages and documentation. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25, New York, NY, USA, 2010. ACM.

[64] H. Melo, R. Coelho, and C. Treude. Unveiling exception handling guidelines adopted by java developers. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 128–139, Feb 2019.

[65] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, pages 120–, Washington, DC, USA, 2000. IEEE Computer Society.

[66] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa. Improving developers awareness of the exception handling policy. In *SANER*, 2018.

[67] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa. Improving developers awareness of the exception handling policy. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 413–422, March 2018.

[68] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing api usages in large source code repositories. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 646–651, Nov 2013.

[69] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *IEEE Software*, 23(4):76–83, 2006.

[70] A. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. Nguyen, J. Al-Kofahi, and T. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 69–79, June 2012.

[71] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.

[72] T. Nguyen, P. Vu, and T. Nguyen. Personalized code recommendation. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 313–317, 2019.

[73] T. Nguyen, P. Vu, and T. Nguyen. Personalized code recommendation. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 313–317, 2019.

[74] T. Nguyen, P. Vu, and T. Nguyen. Recommending exception handling code. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 390–393, 2019.

[75] T. Nguyen, P. Vu, and T. Nguyen. Code recommendation for exception handling. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1027–1038, New York, NY, USA, 2020. Association for Computing Machinery.

[76] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM.

[77] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[78] T. T. Nguyen and T. T. Nguyen. Persona: A personalized model for code recommendation. *PLOS ONE*, 16(11):1–27, 11 2021.

[79] 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)*, pages 795–800, 2015.

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

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

[82] 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)*, pages 416–427, 2016.

[83] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen. Learning api usages from bytecode: A statistical approach. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 416–427, New York, NY, USA, 2016. Association for Computing Machinery.

[84] T. T. Nguyen, P. M. Vu, and T. T. Nguyen. An empirical study of exception handling bugs and fixes. In *Proceedings of the 2019 ACM Southeast Conference*, ACM SE '19, page 257–260, New York, NY, USA, 2019. Association for Computing Machinery.

[85] T. T. Nguyen, P. M. Vu, and T. T. Nguyen. An empirical study of exception handling bugs and fixes. In *Proceedings of the 2019 ACM Southeast Conference*, ACM SE '19, page 257–260, New York, NY, USA, 2019. Association for Computing Machinery.

[86] T. T. Nguyen, P. M. Vu, and T. T. Nguyen. Api misuse correction: A fuzzy logic approach. In *Proceedings of the 2020 ACM Southeast Conference*, ACM SE '20, page 288–291, New York, NY, USA, 2020. Association for Computing Machinery.

[87] J. Oliveira, N. Cacho, D. Borges, T. Silva, and F. Castor. An exploratory study of exception handling behavior in evolving android and java applications. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, SBES '16, page 23–32, New York, NY, USA, 2016. Association for Computing Machinery.

[88] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

[89] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[90] L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSp Magazine*, 1986.

[91] M. M. Rahman and C. K. Roy. On the use of context in recommending exception handling code examples. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 285–294, 2014.

[92] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[93] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[94] M. P. Robillard and G. C. Murphy. Designing robust java programs with exceptions. *SIGSOFT Softw. Eng. Notes*, 25(6):2–10, Nov. 2000.

[95] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio. Understanding the exception handling strategies of java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 212–222, New York, NY, USA, 2016. Association for Computing Machinery.

[96] H. Shah, C. Görg, and M. J. Harrold. Visualization of exception handling constructs to support program understanding. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, page 19–28, New York, NY, USA, 2008. Association for Computing Machinery.

[97] S. Shen, B. Hu, W. Chen, and Q. Yang. Personalized click model through collaborative filtering. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, page 323–332, New York, NY, USA, 2012. Association for Computing Machinery.

[98] X. Song, B. L. Tseng, C.-Y. Lin, and M.-T. Sun. Personalized recommendation driven by information flow. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, page 509–516, New York, NY, USA, 2006. Association for Computing Machinery.

[99] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[100] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 283–297, Riverton, NJ, USA, 2013. IBM Corp.

[101] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Fuzzy set-based automatic bug triaging: Nier track. In *ICSE*, 2011.

[102] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *ESEC/FSE*, 2011.

[103] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Washington, DC, USA, 2009.

[104] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM.

[105] 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 *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 319–328, May 2013.

[106] J. Wang, Y. Yang, S. Wang, C. Chen, D. Wang, and Q. Wang. Context-aware personalized crowdtesting task recommendation. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[107] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[108] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 419–431, New York, NY, USA, 2004. ACM.

[109] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, Berlin, Heidelberg, 2005.

[110] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[111] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 334–345. IEEE Press, 2015.

[112] I. H. Witten and T. Bell. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *Information Theory, IEEE Transactions on*, 37(4):1085–1094, Jul 1991.

[113] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.

[114] H. Zhong and H. Mei. Mining repair model for exception-related bug. *Journal of Systems and Software*, 141:16 – 31, 2018.

[115] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015. IEEE Press.

[116] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343, Berlin, Heidelberg, 2009. Springer-Verlag.