Machine Learning as a Mean to Uncover Latent Knowledge from Source Code

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Ervina Çergani, M.Sc.

geboren in Tirana (Albania).

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr. Christoph Bockisch
Datum der Einreichung:	3. Juni 2019
Datum der mündlichen Prüfung:	12. Juli 2019

Erscheinungsjahr 2019

Darmstädter Dissertationen
D17

.

Çergani, Ervina : Machine Learning as a Mean to Uncover Latent Knowledge from Source Code
Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUprints: 2020
URN: urn:nbn:de:tuda-tuprints-116586
URL: `https://tuprints.ulb.tu-darmstadt.de/id/eprint/11658`

Tag der mündlichen Prüfung: 12.07.2019

Preface

I tend to always question even minor things in life and try to find the answers from simple facts. Maybe this is why I became interested into data analysis. The idea of finding new facts and giving value to simple data really fascinates me. I happily accepted the offer to join the Software Technology Group at TU Darmstadt as a PhD student, which allowed me to expand my knowledge into the software engineering domain and provided me with another kind of data worth exploring. What followed were years of continuous learning, hard work, and many many ups and downs. However, these last years have been a journey that I have not undertaken alone. In the following, I would like to express my deep gratitude to all the people that accompanied me in this journey.

First of all, I want to say a big thank you to my supervisor Prof. Dr.-Ing. Mira Mezini for offering me the opportunity to do my PhD thesis at her group and under her supervision. Thank you for your support during my years of PhD and your research guidance over this path. Thank you for your constructive feedback and for making this thesis possible, even though the path has not always been straight.

Other people that helped me for designing the research path and that I am very thankful for are Dr.-Ing. Sebastian Proksch with whom I collaborated very closely during the first year of my PhD, and Prof. Dr. Sarah Nadi who joined our *recommenders subgroup* as a postdoc. We shared many ups and downs during this time. I am deeply grateful for your advices and your reliability, even after you had long left to pursue your academic career at the University of Zurich and University of Alberta respectively. Many thanks go also to Prof. Dr. Ulf Brefeld and Dr.-Ing. Sven Amann. Our meetings provided me with ideas, confidence, and insights to tackle obstacles on the path to a successful project, as well as this thesis. Thank you all for your patient and your continues feedback and advices. Your guidance strongly influenced the way I think and work. Furthermore, many thanks go also to Dr. rer. nat. Edlira Kuci for our many long friendly conversations in the office and walkings in the park. Thanks for motivating me every time when I felt the finish line of the PhD seemed too far.

Many special thanks goes for Gudrun Harris, for your understanding and making my PhD life easier through the infinite journey of paper works. Thank you for being always there for me, and solving the funding matter with the best means possible. Thank you for encouraging me in the hardest time and helping me to see the light at the end of the tunnel with your positive attitude. You were always available for a good chat and with your humor ensured that I stay focused on my path.

I would like to thank all my other colleagues from the Software Technology Group that I had the pleasure to get to know during my PhD journey: Prof. Dr. Guido Salvaneschi, Dr. Andi Bejleri, Dr.-Ing. Ben Hermann, Dr. Ingo Maier, Dr. Johannes Lerch, Dr.-Ing. Joscha Drechsler, Dr. rer. nat. Lars Baumgärtner, Dr.-Ing. Michael Eichberg, Dr.-Ing.

Mohamed Aly, Dr.-Ing. Oliver Bracevac, Dr. rer. nat. Sebastian Erdweg, Dr.-Ing. Sven Amann, Dr.-Ing. Sylvia Grewe, Dr. Ralf Mitschke, Aditya Oak, Anna-Katharina Wickert, Daniel Sokolowski, David Richter, Dominik Helm, Felix Weirich, Florian Kübler, Jurgen van Ham, Leonid Glanz, Manuel Weiel, Matthias Eichholz, Michael Reif, Mirko Köhler, Nafise Eskandani Masoule, Pascal Weisenburger, Patrick Müller, Ragnar Mogk, and Sven Keidel. Thank you for all the diversity you brought over the years.

Finally and most importantly, I want to thank my family and my friends. First of all, I thank my parents for their unconditional love and trust since the first day I was born, and for supporting me throughout school and university despite of the difficulties life brought in our path. Thanks to you I learned that the impossible becomes possible, if we really want something and work hard for achieving it. I'm grateful that you endured it to build the foundation of my education and that you kept supporting me until I managed to stand on my own feet. A special thank you goes to my dearest grandparents, who with care and love taught me as a child how important education is in someone's life, and how to put priorities in life. Unfortunately, you were not still physically around to celebrate my graduation with me, but you certainly were in my heart. You have shown me how important it is to care for each other as a family and I will always strive to be the good hearted, honest, and generous person that you taught me to be. Many special thanks go of course to my only and best sister. Whenever I need a hand or an honest second opinion for a decision, I know that you will always be there for me. I like that we do not always agree on everything, because in this way we perfectly complement each other, and I'm very grateful for having you in my life. Thank you to all my family for your unconditional love and support in every step I took. Thank you for being my strength in the most difficult moments. Thank you for believing in me, for being patient with me and for always being there for me despite of the geographical distance. Last but not least, I would like to thank all my friends I got to know before and during the PhD journey. I am deeply grateful to have known every single one of you, for excepting me for who I am and for having shared so much with me. I know that they say good friendships are the ones that last longer, and most of you have perfectly reflected it over the years we have shared.

Abstract

Becoming increasingly complex, software development relies heavily on the reuse of existing libraries. Such libraries expose their functionality through Application Programming Interfaces (APIs) for developers to interact with, as effective means for code reuse. However, developers using an API must be aware of how to efficiently and correctly use it in their development tasks in order to deliver simple, clear, comprehensive and correct software. To assist developers work with APIs more efficiently, a family of developer-assistance tools known as Recommender Systems for Software Engineering (RSSEs) have shown to be useful in increasing programmers' productivity. Applications of RSSEs are based on learning API usage patterns by analyzing source code. In reaction to this, many approaches have been proposed for learning API usage patterns from code repositories. However, a major challenge in these approaches is the discovery of latent knowledge in source code. Current approaches heavily rely on program analyses that predefine the learning process, and then use different algorithms to aggregate the detailed information extracted from source code.

On this thesis, we aim to redirect the focus on using advanced machine learning tools to uncover latent knowledge in source code. Machine learning algorithms are known to use general input formats, are fully automated and work well across different domains. Therefore, to investigate the advantages of machine learning approaches and their potential in software engineering, we consider two different dimensions. *First*, we use the same program analyses as used by a state of the art method call recommender [123], and investigate if replacing the existing learning approach (canopy clustering) with a more powerful machine learning algorithm (Boolean Matrix Factorization - BMF), discovers additional knowledge that was not possible with the previous approach. We find that BMF is indeed able to automatically discover the number of clusters to represent the object usage space, and identifies corner cases (noise) in the data, while reducing model size and improving inference speed without compromising prediction quality. *Second*, we use an event stream mining algorithm that can automatically learn different code representations (pattern types), without complex domain knowledge needed to encode a-priori. We evaluate the quality of the learned patterns on the application context of misuse detection, and compare its performance with five state of the art misuse detectors. Our evaluation results show that the patterns learned perform better in terms of precision by ranking true positives higher in the top findings, and in terms of recall by being able to detect more misuses in the source code.

Our results show practical evidence of the positive impact that machine learning tools can bring to the field of software engineering, in terms of automatically discover latent knowledge in source code, and their comparability (or even better) performance with respect to state of the art approaches.

Zusammenfassung

Die Softwareentwicklung wird immer komplexer und hängt stark von der Wiederverwendung vorhandener Softwarebibliotheken ab. Solche Bibliotheken stellen ihre Funktionalität über APIs (Application Programming Interfaces) zur Verfügung, mit denen Entwickler interagieren können, um Code effektiv wiederzuverwenden. Entwickler, die eine API verwenden, müssen jedoch wissen, wie sie diese effizient und korrekt in ihren Entwicklungsaufgaben verwenden können, um einfache, klare, umfassende und korrekte Software bereitzustellen. Um Entwicklern die effizientere Arbeit mit APIs zu erleichtern, hat sich eine Reihe von Tools zur Entwicklerunterstützung, die als Recommender Systems for Software Engineering (RSSEs) bekannt sind, als nützlich erwiesen, um die Produktivität von Programmierern zu steigern. Anwendungen von RSSEs basieren auf dem Erlernen von API-Nutzungsmustern durch Analyse des Quellcode. Als Reaktion darauf wurden viele Ansätze vorgeschlagen, um API-Verwendungsmuster aus Code Repositorys zu lernen. Eine große Herausforderung bei diesen Ansätzen ist jedoch die Entdeckung latenten Wissens im Quellcode. Gegenwärtige Ansätze stützen sich stark auf Programmanalysen, die den Lernprozess vordefinieren und dann verschiedene Algorithmen verwenden, um die aus dem Quellcode extrahierten detaillierten Informationen zu aggregieren.

In dieser Arbeit wollen wir den Fokus auf die Verwendung fortschrittlicher maschineller Lernwerkzeuge lenken, um latentes Wissen im Quellcode aufzudecken. Es ist bekannt, dass Algorithmen für maschinelles Lernen allgemeine Eingabeformate verwenden, vollständig automatisiert sind und in verschiedenen Bereichen gut funktionieren. Um die Vorteile von Ansätzen des maschinellen Lernens und ihre Potenziale in der Softwareentwicklung zu untersuchen, betrachten wir daher zwei verschiedene Dimensionen. *Zuerst* verwenden wir dieselben Programmanalysen wie ein letzte Stand der Technik Methodenaufrufempfehlung [123] und untersuchen, ob der vorhandene Lernansatz (Canopy Clustering) durch einen leistungsfähigeren Algorithmus für maschinelles Lernen (Boolean Matrix Factorization - BMF) ersetzt wird, entdeckt zusätzliches Wissen, das mit dem vorherigen Ansatz nicht möglich war. Wir stellen fest, dass BMF tatsächlich in der Lage ist, die Anzahl der Cluster zur Darstellung des data Objekten automatisch zu ermitteln und Eckfälle (Rauschen) in den Daten zu identifizieren, während die Modellgröße reduziert und die Inferenzgeschwindigkeit verbessert wird, ohne die Vorhersagequalität zu beeinträchtigen. *Zweitens* verwenden wir einen Event-Stream-Mining-Algorithmus, der automatisch verschiedene Codedarstellungen (Mustertypen) lernen kann, ohne dass komplexe Domänenkenntnisse für die a-priori-Codierung erforderlich sind. Wir bewerten die Qualität der erlernten Muster im Anwendungskontext der Missbrauchserkennung und vergleichen ihre Leistung mit fünf letzte Stand der Technik Missbrauchsdetektoren. Unsere Bewertungsergebnisse zeigen, dass die erlernten Muster in Bezug auf die Präzision besser abschneiden, indem echte Positive in den Top-Ergebnissen höher eingestuft wer-

den, und in Bezug auf den Rückruf, indem mehr Missbräuche im Quellcode erkannt werden können.

Unsere Ergebnisse zeigen praktische Beweise für die positiven Auswirkungen, die Tools für maschinelles Lernen auf das Gebiet der Softwareentwicklung haben können, indem sie automatisch latentes Wissen im Quellcode entdecken und deren Vergleichbarkeit (oder sogar bessere Leistung) in Bezug auf modernste Ansätze.

Contents

1 Introduction

Over the last few decades, software has become an essential part of our everyday life: mobiles, TVs, computers, everything runs on software. Despite the continuous advance in state of the art, software development remains a challenging and knowledge-intensive activity, because software systems become increasingly complex. At the same time, developers are continuously introduced to new technologies, components and ideas. In order to keep up with the market speed and the need for introducing new software, the software development process is mainly based on reusing existing software components [18, 141, 142], referred to as software libraries. Rather than implementing new systems from scratch, developers look for, and try to integrate into their projects, libraries that provide functionalities of interest. Such libraries expose their functionality through Application Programming Interfaces (APIs) for developers to interact with, as effective means for code reuse. However, developers using an API must be aware of how to efficiently and correctly use it in their development tasks in order to deliver simple, clear, comprehensive and correct software.

A proactive approach to assist developers work with APIs more efficiently are a family of developer-assistance tools, referred to as Recommender Systems for Software Engineering (RSSEs) [131, 133]. The key idea behind RSSEs is to automatically obtain information items estimated to be valuable for a software engineering task in a given context and to provide them to developers, often directly in their Integrated Development Environments (IDEs). RSSE tools have shown to be useful in assisting developers during their development time as means to increase programmers' productivity [131, 133].

Applications of RSSEs include: code completion [21, 123, 103], code search [178], and bug or anomaly detection [158, 110, 91, 157]. Code completion can be divided into single completion at a time (method completion, parameter completion etc.), and code snippets completion. *Single completion* is heavily used by developers to decide which code element to use next given the current context. While traditional code completion systems only exploit the type system and propose an alphabetically sorted list of all possible code elements in focus, intelligent code completion systems propose relevant code elements by comparing the editor content to code patterns extracted by analyzing large repositories. *Code snippet completion* on the other hand, assist developers by recommending complete code snippets containing multiple code elements, while also comparing the editor content with the learned code patterns. *Code search* engines search through different available sources, such as Google Code[1] and Stack Overflow[2], to provide developers within the IDE code examples that illustrate the use of code elements of interest. *Misuse detectors* analyze the code editor written by the developer and if a violation is identified with

[1] https://code.google.com

[2] https://stackoverflow.com

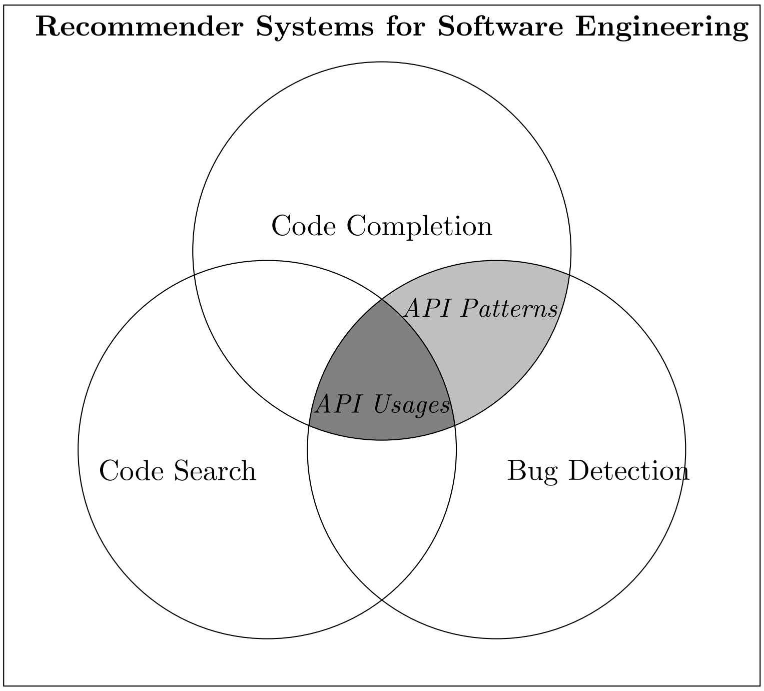


Figure 1.1: RSSE applications based on API usages

respect to API specifications or some mined patterns, the code is marked as a potential anomaly.

All the above mentioned RSSE tools, are based on API usages to provide recommendations to developers within their IDE environment. The only difference is that while code search tools propose to developers API usage examples as extracted from available sources, code completion and anomaly detection tools first analyze the extracted API usages in order to learn API patterns before outputting their final recommendations to developers (Figure 1.1).

A source code snippet that combines one or more API elements in order to accomplish some task, is referred to as an *API usage*. An *API pattern* encodes a set of API elements that are frequently used together (similar API usages), optionally complemented by constraints like the order in which those API elements must be called.

At the same time, the number of open-source projects committed in public version-control systems (VCS), such as GitHub[3] and BitBucket[4], has tremendously increased over last decades, making source code data easily accessible. Given the above facts, learning API patterns is a topic of much interest. In reaction to this, researchers have proposed many approaches to learn API usage patterns from code repositories [132].

The focus of this thesis are API usage pattern learning approaches based on source code for the application domain of code recommenders in software engineering, as shown

[3] https://github.com

[4] https://bitbucket.org

in Figure 1.1, within the context of code completion and misuse detection.

1.1 Problem Statement

As illustrated in Figure 1.1, API usages are the core information in many RSSE tools. They define the source code representation that serves as input information to different API pattern learning approaches within the application context of code completion and misuse detection. However, a major challenge in these approaches is the discovery of latent knowledge in source code. Current approaches heavily rely on specialized hand-crafted domain knowledge to define the source code representation, which in most of the cases also predefines the learning process. Moreover, the learning process is based on algorithms that mainly aggregate the detailed information extracted from source code, instead of uncovering new knowledge automatically. In such approaches, *domain knowledge* represent program analyses performed by the researchers on code repositories, such as domain knowledge feature extraction [123], or the encoding of data and control-flow dependencies into the source code representation [13].

In this thesis, we discuss the limitations of such approaches and investigate the potential of advanced machine learning algorithms to automatically uncover latent knowledge from code representations that encode less semantic knowledge. Our main focus are code completion and misuse detection tools based on source code that use simultaneously *program analyses* and *learning algorithms* to generate their recommendations.

1.1.1 Program Analyses

Many program analysis efforts are invested in order to turn raw code into a sufficiently *interpreted format* that can be further processed. For example, the source code has to be parsed, commits have to be aggregated, and software has to be abstracted into dependency graphs. In the following, we present some of the main program analyses used to convert source code into interpretable formats and discuss their respective limitations:

Program slicing is a type of analysis intended to identify the parts of a program that may affect, or be affected by, the values computed at some point of interest [151]. Although static slicing and its variants are conceptually appealing techniques, they suffer from practical limitations. *First*, computing slices can be expensive [162], and pragmatic considerations may require lower-precision data-flow analyses [152]. *Second*, because a statement is often transitively dependent on many other statements, slices are often very large [162].

Static analysis examine source code statically, as it is written in the code editor before the program is run. Due to the complexity of code, *static analysis* can produce infeasible call sequences. For example by extracting call sequences from both `if` and `else`-clauses [134, 143].

Dynamic analysis collect information as the program executes. Therefore, they heavily depend on the availability and quality of test cases for an executable system. As such, they cannot be applied to incomplete code or to code that cannot be executed [39, 166].

Despite of their limitations, program analyses have many advantages and might reveal information that is not clearly expressed in source code. However, such analyses also comes with some costs when used exclusively to uncover latent knowledge directly from source code:

- A human expert is required for designing such analyses. These analyses are usually developed iteratively: as the results of some analyses being examined, other analyses need to be introduced to either broaden or limit the results that are already identified. Such iterative analyses are developed through manual inspection or empirical analysis, introducing a human burden [133].
- Generalizing to other programing languages is much more difficult, as the analyses need to be implemented differently for every language [150].
- In some cases, the designed analyses can only be used for one specialized task. For example, given a known bug, a human expert is expected to develop program analysis to find occurrences of similar bugs elsewhere in the project, or other projects (i.e. FindBugs [53]).

1.1.2 Learning approaches

Over the years, a variety of approaches have been used by researchers to learn API patterns from source code. In the following, we give a brief overview of these approaches.

Collaborative filtering operates on uncompressed versions of unit data. Such examples include nearest neighbors [21]. The fundamental idea is to recommend to users code elements that have been used by similar users in similar contexts.

Content-based filtering calculates a set of elements that are most similar to other elements already seen in the source code. These approaches compare the content of already used elements (i.e. method calls), with new elements that can potentially be recommended. Concrete applications include *clustering* which groups similar elements together, and *statistical-based methods* that calculate a probability distribution of elements based on other elements already seen in the source code [123].

Data mining finds and summarizes patterns in some structure, and those patterns represent how, in the past, users have explored that structure. Concrete algorithms include *subgraph mining* [13] which uses graph representations, *subsequence mining* [178] using a strict order representation, *frequent item-set mining* finds sets of elements that are frequently called together [66], and *Finite State Automata* (FSA, [121]) calculates a transition function between different states of a program.

The above mentioned approaches are mainly used in the literature to aggregate the detailed information collected by program analyses, instead of automatically discovering latent knowledge from source code. Instances of such program analyses information include the set of features extracted from source code [21, 123], or a predefined code structure which encodes data and control-flow dependencies [13].

At the same time, general purpose machine learning algorithms use simple input formats that are natural, general, fully-automatic, and work well across different tasks and programming languages. We see the following advantages by using advanced machine learning approaches:

- The human expert burden is significantly reduced, since machine learning offers general purpose algorithms not bounded within a specific domain context.
- Generalizing to other programming languages might be as simple as replacing the parser for the new language and using the same traversal algorithm.
- The focus is mainly in automatically discovering latent knowledge not obvious in source code, instead of aggregating the detailed information extracted through program analysis [123, 13].

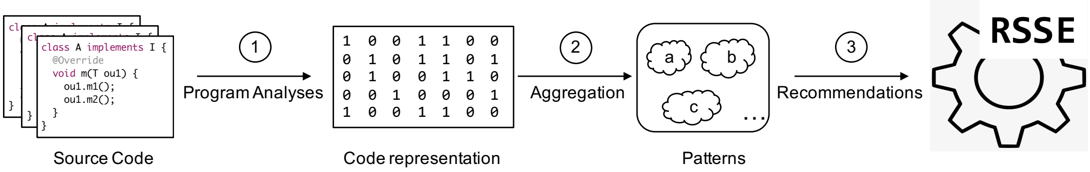
Learning from large available datasets using advanced machine learning approaches has transformed a number of areas such as natural language processing [60], computer vision [135], and recommendation systems [61]. Prominent examples include, but are not limited to automatic machine translation systems, such as Google Translate[5] that learns from existing documents to translate sentences from one natural language to another. Accurate face detection services is another example of successful learning from a large dataset of images. Given the overwhelming success of machine learning in a variety of application domains, we want to bring the same benefits to software engineering. For this purpose, in this thesis, we adapt existing machine learning approaches to the application domain of RSSE, and empirically evaluate how they compare to existing approaches.

1.1.3 Thesis Focus

Figure 1.2 illustrates the general pipeline of code recommendation and misuse detection approaches from the literature. In the *first step*, some program analyses (slicing, static, dynamic) are applied on the source code to transform it into some representation (a set of features, data and control-flow). As described in Section 1.1.1, usually this representation includes some domain specific information extracted from source code which require a domain expert knowledge. In the *second step*, learning algorithms are applied on the code representation, which as described in Section 1.1.2, usually aggregate the detailed information extracted in the previous step in some patterns representation. *Finally*, the learned patterns are inputted to the recommender engine to generate relevant recommendations to the developer, or for detecting violations in source code.

The focus of this thesis lies on the second step of the pipeline in Figure 1.2, where we aim to shift from approaches that simply aggregate the inputted (known) information extracted through domain specific program analyses, into more sophisticated approaches that are able to automatically discover latent (unknown) knowledge from source code. The research question that we address therefore is:

[5] https://translate.google.com



```
class A implements I {
  @Override
  void m(T ou1) {
    ou1.m1();
    ou1.m2();
  }
}
```

```
1  0  0  1  1  0  0
0  1  0  1  1  0  1
0  1  0  0  1  1  0
0  0  1  0  0  0  1
1  0  0  1  1  0  0
```

Figure 1.2: General pipeline of RSSE tools from the literature.

Can RSSE tools profit from machine learning in order to reduce the effort of human-involved program analyses by automatically uncover latent knowledge from the available source code datasets, and still be able to achieve similar or better results compared to the state of the art approaches?

Addressing this question is challenging, because as opposed to other kinds of data, source code represent complex structure and semantics that should be captured during the learning process. Existing naive approaches based on traditional information retrieval methods treat code as natural language text [51]. These approaches however ignore important semantic information of source code and are too imprecise, leading to limited practical applicability.

Therefore, in this thesis we investigate the advantages of machine learning in two dimensions:

First, based on the same program analyses as defined in a state of the art approach (PBN [123]), we investigate if by using a more sophisticated machine learning algorithm we are able to find additional latent knowledge that was not possible to uncover before, and if this knowledge contributes in improving the performance on the current system. For this purpose, we use a *Boolean Matrix Factorization* (BMF) approach. We show that BMF overcomes many of the drawbacks that come with simple clustering approaches, such as automatically discovering the number of clusters to represent the object usage space from source code, and identify corner cases (noise) in the data. To evaluate BMF performance, we use the PBN recommender, which is designed as an extensible inference engine for method completion. We replace the originally used canopy clustering with BMF, and compare the performance of both approaches in terms of model size, prediction quality and inference speed.

Second, we use an event stream mining algorithm (episode-mining) that automatically learns different code representations (sequences, sets and partial-orders), with no complex domain knowledge needed to encode a-priori. Designing a code representation that enables effective learning is a critical task that is often done manually for each programming language. The main idea behind episode mining is to represent source code as a stream of events by traversing its abstract syntax tree (AST). This allows a learning model to leverage the structured nature of code, while still considering AST semantics. We use episode mining to learn different API usage pattern types from source code, perform an empirical study to compare the different pattern types based on three pre-defined metrics (Section 4.4.3), build an API misuse detector based on the learned patterns, and analyze how it compares to state of the art misuse detectors in terms of precision and recall.

1.2 Contributions of this Thesis

This work contributes to the area of code recommenders. Our systematic survey of existing API usage pattern mining literature provides an overview on the state of the art in the field. We adapt the Boolean Matrix Factorization (BMF) approach in the application domain of code completion to automatically learn the number of clusters needed to represent an object usage space and identify noise in the data. Our empirical evaluations show that BMF can successfully identify noisy data, which result in reduced model size of the learned patterns and improved inference speed compared to a state of the art recommender (PBN [123]). Furthermore, we introduce a new benchmark (PtBench) which adapts an existing episode mining approach [1] to automatically identify different representations (sequential, partial, and no-order patterns) from source code, perform a qualitative and quantitative empirical comparison between the different representations, and evaluate the mined patterns on the application domain of misuse detection. Our empirical evaluations show that partial-order patterns are a good trade-off for representing source code in the wild. Furthermore, in the application context of misuse detection we get comparable results to state of the art approaches [13, 91, 158, 110, 157], heavily based on program analysis. We make PtBench[6] publicly available to other researchers for exploring additional properties of API patterns, and for building-up other applications based on API usage patterns.

The State-of-the-Art in API-Usage Pattern Mining Approaches Our first contribution is a systematic literature review of existing work in API-usage pattern mining. In order to push for more advanced machine learning approaches in API-usage pattern learning, first we need to understand how existing approaches are built, and what their current capabilities and limitations are. First, we introduce the large variety of resources from where researchers extract different kind of code snippets. Then, we present an overview of the different code elements analyzed in the pattern learning approaches from the literature. At the end, we present existing approaches used to learn API patterns within the application domain of code completion and misuse detection. We find that most of the approaches are based on static and dynamic analysis, and aggregation algorithms to learn patterns from source code.

Scalability in API Method Call Analytics In our second contribution, we apply BMF approach within the context of intelligent method call completion. Our evaluation results reveal that BMF is an effective approach in reducing the model size of an existing intelligent code completion engine based on canopy clustering, by automatically detecting noise in source code. We extend the BMF approach with a heuristic to further improve pattern detection within the application domain of method call completion. We

[6] http://www.st.informatik.tu-darmstadt.de/artifacts/patternTypes/

perform a systematic empirical evaluation on the effect of BMF on the performance of an intelligent method call completion engine based on three metrics: model size, inference speed and prediction quality. The evaluations show that BMF reduces model size up to 80% and increases inference speed up to 78%, while not compromising prediction quality. This improvement is due to the ability of the BMF approach to automatically identify noise in the data, and remove that noise from the learned patterns.

A Systematic Comparison of API-Usage Pattern Types In our third contribution, we provide a public benchmark (PtBench) that enables the comparison of different pattern types. So far, we lack systematic studies of the tradeoffs between the different types of patterns in representing source code in practice. Such a comparison with regards to some predefined metrics is indeed challenging, because each approach in the literature uses a different learning technique with configurations specific to its data, a different representation for API usages and patterns, and might even be specifically tight to a particular programing language or input form. For this purpose, we first adapt an existing episode mining algorithm [1] that automatically identifies different code representations (pattern types) by simply transforming source code into a stream of events. Then, we define three metrics on which we base on the comparison of the identified pattern types: expressiveness, consistency and generalizability. At the end, we perform an empirical study that compares the pattern types based on the defined metrics. Some of our main findings are: *(1)* the three pattern types don't differ in terms of pattern size and number of API types involved; *(2)* partial-order mining finds additional patterns missed by sequence mining, which are used across repositories; *(3)* sequential and partial-order mining encode important order information for representing correct co-occurrences of code elements in real code. Our findings help in building better applications based on API usages. Furthermore, our benchmark is publicly available to other researchers to evaluate additional metrics on the different pattern types.

The Impact of Order Information in API Method Call Misuse Detection In our fourth contribution, we present EmDetect, a new API-misuse detector as an extension of PtBench. We build EmDetect based on the identified patterns by the episode mining algorithm, and evaluate the quality of the learned patterns in the application domain of misuse detection. Our results show that sequential-order patterns perform better in terms of precision by ranking true positives higher in the top findings, while partial-order patterns perform better in terms of recall by being able to find more misuses in the source code. In general, EmDetect shows comparable results compared to other state of the art approaches [13, 91, 158, 110, 157], in terms of both precision and recall. Our benchmark (PtBench) provides support for two programming languages: C# and Java.

1.3 Publications

Most of the content presented in this thesis have previously been published to software engineering conferences or workshops. This section gives an overview over these publications and the respective parts of this thesis. Parts of the thesis may contain verbatim content of the publications.

Addressing Scalability in API Method Call Analytics [29]. The SWAN'16 paper presents the BMF approach as a mean to reduce the memory cost of an existing intelligent method completion engine (PBN [123]) with no loss in prediction quality (Chapter 3). BMF is evaluated using Eclipse SWT framework (including 44 API types and 190,000 object usages in total), and has shown to improve the inference speed by up to 78% and reduce the model size by up to 80%, without significant changes to the prediction quality, as presented in Section 3.3.3.

Investigating Order Information in API-Usage Patterns: A Benchmark and Empirical Study [30]. The ICSOFT'18 paper presents an empirical investigation of the trade-offs between three pattern types identified from the literature with respect to real code (Chapter 4). Our approach consists of three steps (Section 4.3.2): the transformation of source code into an event stream, the adaptation of an existing general-purpose episode mining algorithm to the special context of pattern mining for software engineering, and filtering the resulting patterns. We define three metrics on which we base the comparison between the identified pattern types: expressiveness, consistency and generalizability (Section 4.4.3). In Section 4.7, we present a set of Implications that we derive from the evaluation results presented in Section 4.6.

EmDetect: On the Impact of Order Information in API Usage Patterns [27]. The paper presents a new API-misuse detector (EmDetect), used as a baseline to empirically evaluate and compare the effect of order information in API patterns, within the application context of misuse detection (Chapter 5). For the sake of completeness, the paper also compares the performance of EmDetect with other state of the art misuse detectors ([13, 91, 158, 110, 157]), in terms of precision and recall (Section 5.4).

1.4 Structure of this Thesis

This thesis is organized as follows. In Chapter 2, we present our survey on existing API usage pattern mining approaches from the literature. Section 2.1 introduces the main terminology used throughout this thesis. Section 2.2 presents the variety of resources from where researchers extract examples of code snippets. In Section 2.3, we present an overview of the different code elements analyzed in the pattern learning approaches from the literature. In order to push for more advanced machine learning approaches in API-usage pattern learning, first we need to understand how existing approaches are built, and what their current capabilities and limitations are. For this, we present in Section 2.4

a systematic literature survey to identify existing work on API-usage pattern learning, and assess and compare the methodologies of respective approaches.

In Chapter 3, we adapt the BMF approach within the application context of recommender systems, to tackle scalability in terms of model size in an existing intelligent code completion engine. Section 3.1 presents the background on how PBN [123] pipeline works, and the motivation behind using BMF. PBN is an extensible pipeline for intelligent method call completion, which we use to compare the previous approach based on canopy clustering with the newly proposed BMF approach. Section 3.2 describes the integration of BMF into the existing PBN pipeline. Section 3.3, first presents the metrics on which we base on the evaluations of the new recommender, and then describes our empirical evaluation results in comparing BMF with canopy clustering. Threats to validity is discussed in Section 3.4, related work presented in Section 3.5, and we conclude this part with a discussion in Section 3.6.

In Chapter 4, we introduce PtBench, our benchmark for learning and comparing different pattern types: sequential, partial and no-order patterns. First, we present related work in mining different pattern types, and empirical studies performed on API usages (Section 4.1). We then discuss how we adapt a general purpose episode mining algorithm to the special context of mining API usage patterns (Section 4.3), then we define the evaluation setup (Section 4.4) for the empirical comparison we perform in Section 4.6. Through automating most of the evaluation process with PtBench, we enable reproducible and comparable results (Section 4.5). To conclude, we derive a set of implications from our empirical comparison that are useful to other researchers working with API usages (Section 4.7). Section 4.8 treats threats to validity, and we conclude this part with a discussion in Section 4.9.

In Chapter 5, we present EmDetect, our new API-misuse detector, used as a baseline to compare the different pattern types within the application context of misuse detection. In Section 5.1, we explain our motivation for building EmDetect. In Section 5.2, we present our detector's violation-detection algorithm and its ranking strategy. Subsequently, we describe the evaluation setup we use to assess EmDetect's performance (Section 5.3), evaluate the detector in terms of precision and recall, and compare it to other detectors from the literature (Section 5.4). Section 5.5 demonstrates the potential for future use of PtBench, where we explain how the benchmark can be further extended with new datasets, new metric definitions, and new applications. To conclude, we discuss the threats to validity of PtBench (Section 5.6), and present an overview of related work (Section 5.7).

In Chapter 6, we conclude this thesis (Section 6.1) and present an outlook to future work (Section 6.2).

2 Background and State of the Art Survey

An *API pattern* encodes a set of code elements that are frequently used together, optionally complemented by constraints like the order in which these code elements must be called. An example of an *usage constraint* is having to call `hasNext()` before calling `next()` on an `Iterator`.

Although using an API can be as simple as calling a function, in practice it is often much more difficult since often interface structures must be accessed by combining interface elements into *usage patterns* [132]. Murphy et. al [97] noted that developers often have difficulties in using unfamiliar APIs during their development tasks. For this reason, developers often search for code examples of API usages to help them complete their tasks. Ideally, development environments should assist developers during their development tasks. Existing development assistance tools have been approached in various ways, e.g., through actively notifying developers about relevant API usage constraints [36], or by recommending correct usages [52, 21, 123].

To make these tools more efficient, and practically usable in developers' everyday tasks, they need to be scalable and provide correct and meaningful, syntactic and semantic code examples within a given context. On the same time, while developing such tools, researchers should also consider that source code itself represent complex structure and semantics, which is not directly visible. For these reasons, the literature offers a large variety of approaches for learning API patterns [132]. However, current tools heavily rely on domain knowledge to extract detailed information from source code, and on using learning algorithms that simply aggregate the detailed information extracted. Even though domain knowledge has many advantages in learning correct patterns, it introduces a huge human burden. In this thesis, we advocate on using advanced machine learning approaches that are able to automatically discover new knowledge from source code, instead of simple aggregation approaches that rely on detailed and specialized domain knowledge encoded a-priori, which also tend to predefine the learning process.

In order to push for more advanced machine learning approaches in API pattern learning, first we need to understand how existing approaches are built, and what their current capabilities and limitations are. This would allow researchers to improve current applications based on API usages, by enhancing the strengths and overcoming the weaknesses of current approaches.

To address these needs, in this part of the thesis, we present a conceptual analysis of the state of the art in API pattern learning approaches. In Section 2.1, we present our main definitions used throughout this thesis. Then in Section 2.2, we introduce the variety of resources from where researchers extract examples of API usages, like API documentation, source code, user's interaction data, and online sites. We find that most of the approaches use published source code as their main data source.

In Section 2.3, we present an overview of the different code elements that are considered in different pattern learning approaches. We find that all approaches consider method calls, and in most of the approaches they are considered exclusively. Often, in the literature, method calls are defined as the main block of communication with the APIs, therefore they are attributed as more important by the researchers.

In Section 2.4, we present the results of a systematic literature review to identify existing API pattern learning approaches within the application domain of code completion and misuse detection. We find that existing approaches are mainly based on static and dynamic analyses, and aggregation algorithms to learn patterns from source code. We qualitatively compare the descriptions of 32 existing approaches, and identify their conceptual capabilities and shortcomings.

The terminology and the approaches presented in this part are helpful to understand the contributions of this thesis. The conceptual analysis of the state of the art in API-pattern learning approaches motivate our work presented in the following parts of the thesis.

2.1 Terminology

Application Programming Interfaces (APIs) provide abstraction mechanism that enable complex functionality to be used by client programs. However, such abstractions do not come for free, because understanding how to use an API in practice can still be difficult. Therefore, researchers have developed different approaches for learning API patterns to provide to developers during their development time. An *API pattern* encodes a set of basic *code elements* that are frequently used together, optionally complemented by constraints like the order in which they must be called. We call such constraints *API usage constraints*, and they are specific for a given API. For example, calling `hasNext()` before calling `next()` on an `Iterator`, in order to avoid a `NoSuchElementException` at runtime. Approaches that automatically identify patterns from a given codebase are referred to as *API-pattern learning approaches.* Such approaches use as input an abstract representation of source code responsible to accomplish some task, referred to as an *usage.* A given usage that uses one or more API code elements is known as an *API usage.* An *API usage pattern* is an API usage that contains all of the code elements from a given pattern, satisfying the respective *usage constraints* enforced by the pattern.

2.2 Sources of API Usages

The quality of the input data is one of the main factors for building successful tools [5]. Therefore, deciding on the sources from where to get this data is one of the most important decisions while building RSSE tools. The decision should consider the main purpose of the tool to be build, and what kind of support it is designed to offer.

API-pattern learning approaches extract API usages from a large variety of sources, such as source code repositories, API documentations, user's interaction data, and online sites. For every such data source, we present some of the main platforms that make them accessible for researchers, and explain the corresponding usage scenarios.

2.2.1 Source Code Repository

Source code is publicly available in many repositories of open-source projects. After decades of development, software repositories accumulate many source files that illustrate API usages. In recent years, it has been a hot research topic to mine specifications from such source files, where a mined specification defines the legal sequences or the invariants (e.g., preconditions) for calling APIs. A *source-code repository* is a web hosting facility where a large amount of source code for software is kept, either publicly or privately. They are often used by open-source software projects and other multi-developer projects to handle various versions. Some of the main services hosting source code of open-source projects include: Assembla[1] hosting more than $100K$ client projects. BitBucket hosting $900K$ teams and $5M$ developers on its platform. GitHub in June 2018 reports hosting over $57M$ repositories (including $28M$ public repositories) and $28M$ users, making it the largest host of source code in the world. LaunchPad[2] in June 2018 hosted more than $40K$ projects. SourceForge[3] in March 2014 claimed to host more than $430K$ projects and had more than $3.7M$ registered users.

Despite of the many other alternatives of available sources of API usages, source code seems to be the most commonly used one for research on APIs. It represents the most up-to-date artifact and therefore the most reliable data source. Source code provides a rich and structured source of information upon which researchers can rely on for training their models, and provide useful recommendations to software developers. Furthermore, since programming lies at the heart of software development, it is no surprise that recommendation systems based on source code analysis draw such an impact. Recommendation systems that rely on source code repositories as their input data, provide support for tasks such as how to use a given API [22], provide hints on things missing from the code [97], suggest how to reuse [80], [169] or correct an existing code.

Even though there are many RSSE tools that rely on source code as their main input data source, they analyze code mainly based on *static* and *dynamic* analyses. MAPO [178] *statically analyze* source files to extract API usages. Due to the complexity of code, static approaches can produce infeasible call sequences, for example by extracting call sequences from both `if` and `else`-clauses. However, it is easier for static analysis to extract all the API usages from a piece of source code compared to other types of analyses. Nguyen *et al.* [112] statically analyze *byte-code* to extract API usages. It is simpler to analyze byte-code than source code, but analyzing byte code has its unique challenges. For example, Meng and Miller [82] complain that byte code can have non-code bytes, missing symbols, and overlapping instructions, which complicate the analysis. Ammons *et al.* [14] execute source code with various input values to analyze *execution traces*. Although the API usages extracted from execution traces are accu-

[1] www.assembla.com
[2] https://launchpad.net
[3] https://sourceforge.net

rate, it can lose some other API usages occurring in source code, due to the difficulty to prepare sufficient test cases. Furthermore, scaling dynamic analyses techniques has proven difficult in industrial programs, because they require perfect traces, and do not work well in situations where only imperfect traces are available.

Each of the above mentioned types of code analyses come with their own advantages and disadvantages, depending on the main purpose of the respective tools using them. In the next chapters of this thesis, we use static analyses, since we are interested in extracting static API usages as they are written in the IDE editor.

2.2.2 API Documentation

Traditional API documentation provide the most direct and intuitive reference in learning how to use APIs correctly. Documentation complements the API by providing information not obvious from the API syntax. However, with the fast evolution of software, many of such documentation get fast outdated and are not considered a trustful resource anymore. Due to the limitation of development time and schedule, many of the newly implemented API specifications are missing from proper documentation. Moreover, software developers would rather write code than documentation. For instance, Saied *et al.* [139] carried out an observational study on API usage constraints and their documentations. The results show that three out of four constraint types, from 79% to 88% usage constraints are not documented. On the other hand, Zhou *et al.* [182] found that more than half (51.2%) of all specifications mined in the Java Class Library are incorrectly documented.

However, API documentation are still considered an important source of information, especially for newly created and unpopular APIs. Thummalapenta *et al.* [148] found that even popular libraries have unpopular API classes. Their results do not indicate that unpopular APIs are useless, since with the evolution of software, unpopular APIs can become popular. For example, the latest API library can implement many new APIs. Furthermore, Zong *et al.* [176] found that most of APIs do not have much client code, so we have to learn their usages from other data sources. Newly released APIs, typically cannot be found in client code, but later they can become popular. If many APIs are unpopular or do not have sufficient client code, researchers usually consider other sources such as their respective documentations (i.e., Doc2Spec [180]).

Furthermore, in the recent years many approaches have been developed that facilitate the maintenance of API documentation, by automatically generating and maintaining them (e.g., [146], [93], [25]). These approaches are useful in keeping API documentation up-to-date, and therefore making them a good reference in learning APIs.

2.2.3 Interaction Data

Interaction data is considered the new source of information in software engineering. It refers to the data that captures and describes the interactions (i.e., edits) of developers with artifacts (i.e., source code entities) using tools (i.e., IDE). These actions are usually performed within a context, for example a specific task. Interaction data is used to investigate developers' behaviors, their intentions, their information needs, and problems encountered, providing new possibilities for precise recommendations. Interaction data creates the possibility to get access to more fine-grained change information or to activities that describe the in-IDE development process. The idea is that single interactions such as code changes, allow for a better understanding of a developer's work and thus for more fine-grained and precise recommendations.

Interaction data typically involves four types of data: *(1) Interactions*, are the actions taken by a developer, for example changes to code entities. *(2) Artifacts*, are the entities the developer is interacting with, for example source code entities, issue reports, or email documents. *(3) Tools*, are software applications developers use during their work, for example the IDE, the issue tracking system, or the email client. *(4) Contexts*, are circumstances in which the developer is performing an interaction, for example tasks a developer is working on or the issues being encountered.

However, very few datasets exist of interaction data. since collecting them is a lot harder compared to other artifacts. Singh *et al.* [144] tracked activities of almost 200 developers, covering more than $30K$ hours of active development time. They make use of developer activity logs to analyze sequences of developer actions, such as navigation and edit actions. Dias *et al.* [37] propose the tool EPICEA that tracks edit related operations in the IDE. The tool also preserves source-code changes on a structural level (e.g., method names), but ignores method bodies. Their data set was collected over four months from seven participants.

Recently, many RSSE tools based on interaction data have been proposed. One example is Mylyn [59], which tracks the selections and edits of source code artifacts to filter most relevant artifacts for the current task. Other examples include Robbes *et al.* [130] and Lee *et al.* [64] that use interaction data to suggest reusable pieces of code and predict defects respectively.

2.2.4 Online sites

In the past, there have been many commercial and scientific attempts by researchers (e.g., [74]) to provide web-based search engines for code. Examples include Google Code Search, Koders, Krugle, Sourcerer, and Meobase. However, none of them ever reported significant numbers of users comparable to mainstream search engines. In fact, shortly after they all shut down their code search engine illustrating that developers need some other form of support.

Recently, online sites such as STACKOVERFLOW and GITHUB have shown to be commonly used among developers. They fill the gap between traditional API-documentation and more example-based resources. Moreover, these online sites have also become an important data source for empirical research on software engineering. While existing research was mainly focused around the textual parts of the posts, programmers ask many questions about a specific coding problem when they are stuck with their solution. Their questions often contain incomplete code snippets, which are completed or rewritten by the community. The high heterogeneity of the posts makes it challenging to use them though. Posts mix textual parts, source-code (which itself is often not complete, invalid, or simply does not specify the programming language), and data in formats like `XML` or `JSON`. Ponzanelli *et al.* [116] solve this with an *island grammar* [94] that can be used to parse StackOverflow posts into *heterogeneous AST* (`H-AST`). The grammar supports `Java`, `XML`, `JSON`, stack traces, and text fragments and can be used to transform released StackOverflow data dumps. Tools like Baker [146] can then be applied to recover typing information in these code snippets, making it possible to use fully-qualified references for types and type elements in static analyses, without having to compile the respective snippets.

2.3 Code Elements in API Usage Patterns

A *code element* refers to an instance that can be found in source code, such as: types, methods, exception handling, parameters, iterations, fields, and conditions. To the best of our knowledge, no work systematically defines the variety of code elements considered in learning API usage patterns. This prevents us from assessing which aspects of source code have been addressed or may have been neglected by existing approaches. To improve on this situation, in this section, we present the variety of elements that can be found in source code and how they are analyzed by existing learning approaches.

2.3.1 Object Types

An *object type* is a specific instance of an API type provided within a library. An *API type* itself defines *methods* that represent operations that can be applied on its objects.

Very few approaches (CodeWeb [85]) learn patterns on how API types are reused in practice. This is done by mining existing applications that use the library. Such applications can use libraries in a way that takes into account inheritance relationships, introducing the pattern practice to the developer. For example, application classes that inherit from a particular library class often instantiate another class or one of its descendants. More concretely, applications that inherit from a library class `Widget()` tend to override its member function `paint()`.

Traditionally, such knowledge is represented by examples in library tutorials and/or toy programs. However, as mentioned in Section 2.2.2, not all programs come with such representative examples of reuse. This is particularly true for libraries developed by a company for internal use and libraries developed by the open source community. Therefore, approaches that learn patterns of API types are particularly beneficial to identify such characteristic usage of the library.

2.3.2 Method Calls

Developers frequently need to use method calls they are not familiar with or they do not know how to use. However, *method calls* are the most prominent elements in source code, as they are the primary means of communication between client code and the API. Usually, developers need to know what are the typical invocation scenarios for a given method. To this aim helps the knowledge about the steps required to invoke this method, such as, invoking other methods or manipulating the method's parameters.

To learn about such methods, developers usually resort to sources such as API documentation (e.g., JavaDoc), developers online forums (e.g., Stack Overflow), or other information sources. However, most of these sources provide generic explanation of the method usage syntax, or focus on the method's technical details. Furthermore, using these kind of sources require the developers to frequently switch context between the IDE and web browsers during development time, which might be confusing. In such cases, developers could benefit from short code fragments presenting practical uses of method calls directly in their IDE.

Therefore, it is reasonable for researchers to focus on approaches that automatically learn patterns of method calls, since methods are more frequently called compared to other types of code elements. In fact, from all the approaches that we review (Section 2.4.2), 69% analyze exclusively usages of method calls.

2.3.3 Exception Handling

Exceptions serve as a mean for APIs to communicate errors to client code. Exception handling allows an error detected in one part of the program to be handled elsewhere depending on the context. For example, when a method does not have enough information to handle ”exceptional” conditions, it ”throws” an exception to a parent method up in the call stack, which contains sufficient context to properly handle the error.

The handling of different errors often depends on the specific API. For example, when initializing a `Cipher` with an externally provided cryptographic key, one should handle `InvalidKeyException`. Another example is resources that need to be closed before use (by calling methods such as `lock()`), which also creates a case of an exception. Also calling `read()` without a preceding `open()` causes an exception. Such guarantees are often implemented by a `finally` block, but also using the `try`-with-resources construct or even respective handling in multiple `catch` blocks.

Unfortunately, exceptions introduce an inter-procedural flow of control that can be difficult to reason about either for human or automatic tools and analysis (e.g., [31, 33]). Failure to correctly handle exceptions lead to security vulnerabilities, breaches of API encapsulation, and any number of safety policy violations. Uncaught exceptions and poor support for exception handling are reported as major obstacles for large-scale and mission control systems (e.g., [9, 23, 26]). Often exceptions are caught trivially (i.e., no action is taken to resolve the underlying error [160]) or the mechanism is purposely circumvented [134].

Buse *et al.* [25] proposes an automatic approach based on symbolic execution and inter-procedural data flow analysis that alerts developers to the presence of ”leaked” exceptions, as well as to the causes of those exceptions. The tool can also be used to automatically generate documentation.

2.3.4 Parameters

In object-oriented programming, a *parameter* defines the input to a method call. Hence, a parameter is a useful and critical element of the method, in performing a specific task. Passing the right parameters in the specified order, is essential to ensure the correct execution of a method.

Bruch *et al.* [21] and Pradel *et al.* [119] show that it is a non-trivial task to choose the right parameter(s) for a method call in an API usage. Zhang *et al.* [172] show that 64% of the method declarations in Eclipse 3.6.2[4] are parameterized, that is, the methods are passed one or more parameters when being called. Besides slowing down the development process, unfamiliarity with parameter usage may even harm the correctness of programs [119]. For example, in statically-typed programming languages, the compiler ensures that method arguments are passed in the expected order by checking the type of each argument. However, calls to methods with multiple equally-typed parameters slip through this check.

Recently, various approaches have been proposed for mining code examples (e.g., [66, 71]), or showing common API call sequences (e.g., [178, 14]). However, few approaches focus specifically on recommending API parameters (e.g., Precise [172]). Precise recommends the kinds of API parameters that are frequently used in practice, but mostly overlooked by existing code completion systems. It is the first automatic technique focusing on parameter recommendation.

2.3.5 Iteration

Iteration is another mean of interacting with APIs, used, in particular, with collections and IO streams. It takes the form of loops and recursive methods. Loops are used often in source code, and many simple tasks (e.g., count, compare pairs of elements, find the maximum/minimum etc.) are implemented as loop structures. Note that respective usage constraints are about (not) repeating (part of) a usage, rather than about the condition that controls the execution.

To the best of our knowledge, Wang *et al.* [156] is the only work that analyzes loop constructs exclusively. Wang *et al.* present an approach to automatically identify the high-level actions of loops in Java methods by abstracting key features from loop code fragments. The approach is focused on analyzing high-level actions implemented by loops. Other approaches [24, 13] analyze loops in correlation with other code elements as well, such as: method calls, conditions, exception handling etc.

2.4 Survey on API Usage Pattern Learning Approaches

We discussed in Section 2.2 that researchers have explored many sources for learning API patterns, such as: source code, documentation, interaction data and online sites. However, most commonly used and up-to-date source are considered source code repositories, where API-usage patterns may be learned through *static analyses* of source code

[4] https://www.eclipse.org

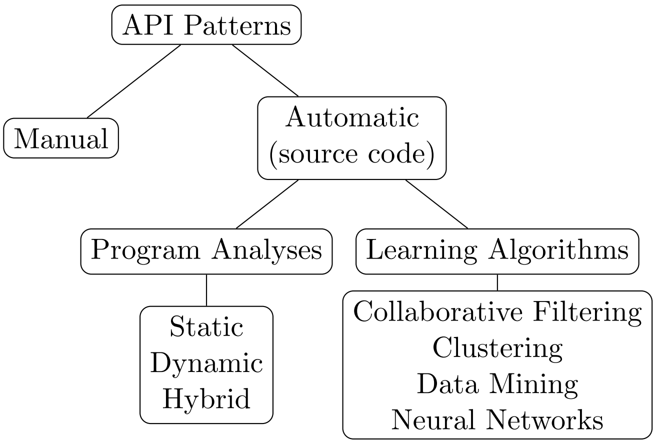


Figure 2.1: API Patterns generation diagram

or binary code, and through *dynamic analyses*, i.e., runtime monitoring or analysis of runtime data, such as execution traces (Section 2.2.1).

Figure 2.1 shows a general diagram on API patterns generation that can be found in the literature. In some cases, API patterns have been *crafted manually* by experts or *inferred automatically* by different learning approaches. However, manually crafting and maintaining API patterns is costly. On the other hand, automatic specification inference (or *mining*) of patterns usually includes two distinguished steps: (1) *Program analysis*, which may, again, be approached both *statically*, e.g., based on source code, code toy examples or documentation, and *dynamically*, e.g., based on traces or logs, or interaction data extracted from the IDE. (2) *Learning algorithms*, that define the pattern abstraction from the information collected in the first step.

Approaches that are based on automatic specification inference are called *API-pattern learning.* In the literature, we find *static pattern learning* that statically mine specifications through static analysis, e.g., [158, 110, 92]; *dynamic pattern learning* that dynamically mine specifications through dynamic analysis, e.g., [120, 73]; and *hybrid pattern learning* that, for example, combine dynamic specification mining with static analysis [121].

To abstract the quantity of collected data through either static or dynamic analysis, researchers use a large variety of learning algorithms to define the learned patterns. In the literature, we find: *collaborative filtering* [21], *clustering* [123], general *data mining* algorithms based on data summarization [13], and *neural networks* [164] that learn a parametrized model from source code.

In this section, we focus on understanding the capabilities and short-comings of existing pattern learning approaches. Therefore, we conduct a systematic literature review to identify existing API-pattern learning and assess their underlying approach, their conceptual capabilities, the type of analysis and algorithm used, and the evaluation setting they were tested in. The focus of our review are approaches that automatically infer API patterns by analyzing source code repositories, so basically learning approaches covered on the right branch of Figure 2.1.

2.4.1 Methodology

We performed a systematic literature survey, starting from a survey of automated API-property inference techniques by Robillard *et al.* [132], 2013 to 2019 proceedings of the ICSE, and 2013 to 2018 proceedings of the FSE and ASE conferences (and their respective co-located events). In this survey we only considered technical track and journal track papers, focusing in this way on fully published results instead of intermediate results. The author of this thesis proceeded as follows:

(1) She manually filtered the proceedings for publications whose title contains one of the keywords: `automatic`, `specifications`, `inference`, `mining`, `learning`, `detect`, `patterns`, `recommender`, `completion`, `bug`, `misuse`, `usage`, `reusable`.

(2) She manually reviewed the title and abstract of filtered publications to identify those that present automatic learning approaches based on source code artifacts.

(3) If a publication presents such an approach, she added all references to other publications that supposedly also present such an approach, to the list of publications to check.

The publications reviewed in this process are classified in two main categories:

(1) *Closely related approaches* to the work presented in the following chapters of this thesis. This category include approaches that:
   - Automatically learn code patterns based on source code data.
   - Simultaneously use program analyses for extracting code artifacts, and mining algorithms for abstracting code patterns from the extracted artifacts.
   - Find applications within the domain context of code recommendation and / or misuse detection.

(2) *Other approaches* also based on automatic knowledge discovery from source code, but that differ from the first category because:
   - They are applied on a different application context, for example automatic test generation [19], code synthesis [115], bug reports summary [181], software artifacts classification [76], clone detection [153].
   - Are not based on code patterns, but instead on API component searching and browsing techniques, and code-example retrieval techniques.

We exclude from our survey: *(1)* approaches based on manually crafted specification (templates) [62, 69], *(2)* approaches based on natural-language specifications extracted from JavaDoc comments, since these approaches do not use source code as their data input [113], *(3)* approaches based on some simple ranking strategies based on software artifacts occurrences [175, 59, 77], *(4)* approaches based solely on program analyses [109, 177], and *(5)* approaches based on code migration [104] that are mainly based on the same principals of language translation.

We reviewed a total of 65 different approaches from the literature, where 51 of them fall in the *closely related* approaches category mentioned above and 14 in the *others* category. We present a detailed description of the *closely related* learning approaches identified in this survey process chronologically by their publication date in Section 2.4.2. For this part, we use the published description and evaluation results of each approach to identify their capabilities, and the program analysis and/or learning algorithm used. We also describe the strategies used to evaluate each approach. In Section 2.4.3, we present a short summary of the *other* approaches based on automatic knowledge discovery from source code artifacts.

2.4.2 Closely Related Learning Approaches

In the following, we describe the reviewed approaches in this category with respect to the following features: the programming languages they support (*Target Language*), the algorithm used to learn patterns (*Mining Algorithm*), the target code elements analyzed to learn the API patterns (*API Elements*), the program analyses used to extract source code artifacts (*Analyses*), the application context for evaluating the learned patterns (*Applications*), and the pattern types learned (*Pattern Type*). For each of the approaches reviewed, we denote with an x if the approach supports the respective feature. Although, there are some exceptions. For example, for the approach presented by Gruska *et al.* [44], we have denoted with *independent* the *Target Language*, since the approach uses a language independent parser for extracting code artifacts. Furthermore, other approaches have no assignments for the feature *API Elements.* This is the case when the respective approach either does not learn patterns of repetitive API elements, for example Lee *et al.* [64] extract general metrics from source file; or when an approach learns patterns that include any possible element from the source code, for example NGSE [51] and White *et al.* [164] treat code as natural language *tokens*, and Liu *et al.* [70] extract code elements as features that occur in source code. Also, Wang *et al.* [156] has no assignments for the feature *Pattern Type*, because the proposed approach does not learn code patterns, but is used instead to automatically generate the high level action associated with loop structures in source code. In the following, the analyzed approaches are presented in more details, and Table 2.1 puts together a summary of this review.

CodeWeb [85] is applied in two C++ frameworks (ET++ and KDE). Items mined by CodeWeb are aggregated at the class level. For example, if any function or method defined in a class A calls a function f, the class as a whole is considered to call the function. CodeWeb is based on association rule mining to learn library reuse patterns, taking into account inheritance hierarchies. For example, application classes that inherit from a particular library class often instantiate another class or one of its descendants. CodeWeb contains the restriction to mine only patterns with one antecedent and one consequent, the algorithm is reduced to mining co-occurring pairs of elements with a given support and confidence. CodeWeb removes patterns stating, for example, that a

Table 2.1: API-Usage Pattern Learning Approaches.

		Mining Algorithm							API Elements						Analyses		Applications			Pattern Type		
Approach	Target Language	Clustering	Statistical	FSA	Item-set	Subsequence	Subgraph	Deep Learning	Method Calls	Iterations	Conditions	Classes	Parameters	Exceptions	Static	Dynamic	Recommendation	Bug Detection	Documentation	Unordered	Sequential	Partial
CodeWeb [85]	C++				x				x			x			x				x		x	
Ammons *et al.* [14]	C			x					x							x		x			x	
Whaley *et al.* [163]	Java			x					x						x	x		x	x		x	
PR-Miner [66]	C/C++				x				x						x			x		x		
RASCAL [80]	Java	x							x						x		x			x		
DynaMine [71]	Java				x				x						x			x		x		
JIST [10]	Java			x					x							x			x		x	
Scenariographer [140]	Java	x							x							x			x		x	
Weimer *et al.* [161]	Java			x					x						x	x		x			x	
Perracotta [166]	C			x					x							x		x			x	
Chronicler [126]	C					x			x						x			x			x	
Jadet [158]	Java			x					x						x			x			x	
Ramanathan *et al.* [127]	C				x	x			x		x		x		x			x			x	
ParseWeb [147]	Java	x				x			x						x		x				x	
Quante *et al.* [125]	Java/C			x					x	x						x		x			x	
Lo *et al.* [72]	Java		x						x							x		x			x	
CBFA [173]	Java/C	x							x						x		x			x		
BMN [21]	Java	x			x				x						x		x			x		
Acharya *et al.* [3]	C					x			x		x					x		x			x	
MAPO [178]	Java	x				x			x						x		x				x	
Alattin [149]	Java				x				x		x				x			x			x	
CAR-Miner [150]	C++/Java				x				x					x	x			x			x	
GROUMiner [110]	Java						x		x	x	x				x			x				x
DMMC [91]	Java		x						x						x		x	x		x		
OCD [40]	Java			x					x	x						x		x			x	
TAUTOKO [34]	Java			x					x							x		x			x	
Gruska *et al.* [44]	independent			x					x						x			x			x	
SpecCheck [101]	Java			x					x							x		x			x	
Tikanga [157]	Java			x					x						x			x			x	
Lee *et al.* [64]	Java	x	x												x			x			x	
Pradel *et al.* [121]	Java			x					x		x				x	x		x			x	
Precise [172]	Java	x											x		x		x			x		
NGSE [51]	Java		x												x		x				x	
Buse *et al.* [24]	Java	x							x	x		x		x	x				x		x	
UP-Miner [154]	Java	x	x			x			x						x		x				x	
SLANG [129]	Java		x						x						x		x				x	
CodingTracker [100]	Java				x				x						x		x			x		
JSMiner [107]	JavaScript						x		x						x			x	x			x
PBN [123]	Java	x	x						x						x		x			x		
MLUP [137]	Java	x							x						x				x	x		
Wang *et al.* [156]	Java	x								x					x				x			
White *et al.* [164]	Java							x							x		x				x	
DroidAssist [111]	Java		x						x						x			x			x	
DeepAPI [46]	Java							x	x						x		x				x	
ApiRec [102]	Java		x						x						x		x			x		
HAPI [112]	Java		x						x						x		x				x	
Salento [96]	Java		x						x							x		x		x		
Liu *et al.* [70]	Java	x						x							x			x			x	
ExampleCheck [174]	Java					x			x		x				x			x			x	
MuDetect [13]	Java						x		x	x	x			x	x			x				x
Focus [108]	Java	x							x						x		x			x		

class that calls a library function on type A must also instantiate this type. The learned patterns are used for API documentation.

Ammons *et al.* [14] uses the k-tail algorithm [16] to learn automata from called sequences of API methods. They first compute a probability FSA (PFSA), and then convert it into a regular FSA by removing the probabilities from the edges while at the same time deleting entirely such edges that are labeled with a probability below a certain threshold. Ammons *et al.* point out that their underlying k-tail algorithm can ignore some sequences that do not fit the learned automata. It is reasonable to ignore some details, since such details are relevant to only specific implementation purposes. Ammons *et al.* implement two different *tracers* for extracting execution traces interactions with an API or ADT (abstract data type): one is a replacement for the C `stdio` library that requires recompiling programs, and the other is a more general executable editing tool that allows arbitrary tracing code. They infer specifications by observing program executions and concisely summarizing the frequent interaction patterns as state machines that capture both temporal and data dependencies. These state machines can be examined by a programmer, to refine the specification and identify errors.

Whaley *et al.* [163] uses multiple finite state-machine (FSM) sub-models to model the interface of a class. In their definition, a sub-model includes a subset of methods that, for example, implement a Java interface. Each state-modifying method is represented as a state in the FSM, and transitions of the FSMs represent pairs of consecutive methods. They use static analyses to deduce illegal call sequences in a program, dynamic instrumentation techniques to extract models from execution runs, and a dynamic model checker that ensures that the code conforms to the model. Extracted models can serve as documentation, or as constraints to be enforced by a static checker. Their system has been run on several large code bases, including the basic Java libraries, and the Java 2 Enterprise Edition library code.

PR-Miner [66] parses functions in C source code to store as items, representing functions called, types used, and global variable accessed. It encodes usages as the set of all function names called within the same function. PR-Miner uses inter-procedural analysis to detect project-specific patterns, and employs closed frequent item set mining, i.e., where there are no sub item sets that are subsumed by larger item sets with the same support. Once identified, the patterns are used to find violations. PR-Miner is evaluated on three C/C++ systems.

RASCAL [80] is a recommendation system that aims to predict the next method that a developer could use, by analyzing Java classes similar to the one currently being developed. RASCAL relies on the traditional recommender technique of *collaborative filtering*, which is based on the assumption that users can be clustered into groups according to their preferences for items. In RASCAL's terminology "users" refer to classes, and "items" refer to methods to be called. The similarity between different classes is based on the methods they call.

DynaMine [71] mines software revision histories to mine common error patterns of method calls for the purpose of bug detection. DynaMine infers usage patterns by mining the change history of source code. The patterns learned by DynaMine are pairwise association rules for methods found in a single source file revision. DynaMine translates the usage pattern mining problem into an item-set mining problem by representing a set of methods committed together into a single file as an item set. DynaMine mines changes at the file level because it only considers files that were actually changed in a given client within some time window. Mining at a finer granularity would likely result in very few patterns. DynaMine uses a pattern filtering phase to "greatly reduce the running time of the mining algorithm and significantly reduce the amount of noise it produces". The experiments are performed on Eclipse and jEdit, two large, widely-used open-source Java applications.

JIST [10] is an automatically approach for extracting temporal specifications of sequences of method calls for Java classes. Given a Java class, and a safety property such as "the exception E should not be raised", the corresponding (dynamic) interface is the most general way of invoking the methods in the class so that the safety property is not violated. JIST first constructs a symbolic representation of the finite state-transition system obtained from the class using predicate abstraction, and then uses algorithms for learning finite state automata and symbolic model checking for branching-time logics.

Scenariographer [140] is an approach for generating class usage scenarios, for example how method sequences of a class can be invoked, which are collected during the execution of a software. The approach employs the notion of canonical sets to categorize method sequences into groups of similar sequences. Scenariographer is evaluated on Java open source programs.

Weimer *et al.* [161] mines temporal specification for bug detection. Their approach is based on the observation that programs often make mistakes along exceptional control-flow paths, even when they behave correctly on normal execution paths. The approach is applied on existing Java programs, which are presented as a set of static or dynamic traces. each of which is a sequence of events. Static traces are generated from the program source code. Dynamic traces are produced by running the program against a workload. Events are taken to be context-free function calls. Mined specifications are finite state machines with events as edges.

Perracotta [166] mines temporal API rules from imperfect traces based on finite state machines. Temporal properties constrain the order of occurrence of program events. For example, acquiring a `lock` should eventually be followed by releasing the `lock`. Perracotta aggregates ordered pairs of API patterns into larger patterns. For instance, from $a \rightarrow b$ and $b \rightarrow c$ one may infer that $a \rightarrow b \rightarrow c$. Perracotta is evaluated in three scenarios, on inferring API rules for Daisy file system, Windows kernel, and JBoss core components. The authors use the inferred properties to validate the program satisfies those properties using static verifiers.

Chronicler [126] is a misuse detector for C. It mines frequent call-precedence relations from an inter-procedural control-flow graph. Typically, these patterns are sequences of method/function calls. Chronicler finds project-specific patterns of method calls. The authors compare the identified protocols with the documented protocols for a given API.

Jadet [158] is a misuse detector for Java. It uses Colibri/ML [67], but instead of only method names, it encodes method-call order and call receivers in usages. Therefore, it first builds a directed graph of a finite state automata with anonymous states whose nodes represent method calls on a given object and whose edges represent control flows. From this graph, it then derives a pair of calls for each call-order relationship, e.g., `m()` $\prec$ `n()`. Each usage is represented by the set of these pairs. These sets of call pairs form the input to the mining, which identifies patterns, i.e., sets of pairs. JADET mines sequential patterns that consist of an ordered pair of API elements (a, b), indicating that the usage of element a should occur before b in a program's execution. JADET finds project-specific patterns.

Ramanathan *et al.* [127] is a misuse detector for C. It statically encodes usages as sets of properties for each variable `v`. Properties are comparisons to literals, e.g., $(\neq, \texttt{null})$, if `v` was checked to be not `null`, argument positions in function calls, e.g., $(\text{arg}(2), \texttt{f})$ if `v` was passed as the second argument to a function `f`, and assignments, e.g., $(:=, \text{res}(\texttt{f}))$ if the `v` was assigned the result of a call to `f`. They derive these properties using an inter-procedural path-sensitive data-flow analysis that gathers predicates at each program point. For each call, Ramanathan *et al.* creates a group of the property sets of the call's arguments. To all groups for a particular function, it applies sequence mining to learn common sequences of control-flow properties and frequent-itemset mining to identify all common sets of all other property types. This approach is designed to detect project-specific patterns of method calls, conditions, parameters.

ParseWeb [147] shows users how to create an object of some target type given an object of another type, by suggesting frequently used method-invocation sequences that can serve as solutions to yield the destination object from the source object. The approach performs static analyses over source code to extract required sequences, and clusters similar sequences using a sequence post-processor. PARSEWEB also sorts the final set of sequences using several ranking heuristics. PARSEWEB is implemented for helping Java code reuse.

Quante *et al.* [125] present a dynamic protocol recovery technique based on object process graphs, a finite representation of the sequences of operations for particular objects extracted from source code or via dynamic analysis. These graphs contain information about loops and the context in which methods are being called. They use an automaton-based approach to transform the extracted graphs to method call sequences. Quante *et al.* evaluate their approach on several Java and C applications.

Lo *et al.* [72] present an approach that can mine patterns of arbitrary length for the purpose of misuse detection in Java programs. It mines statistically significant specifications of the form $\texttt{consequences} \hookrightarrow \texttt{premises}$ from program execution traces, where both the premises and consequences are sets of method calls. It requires that the respective traces contain only calls to methods of the relevant API(s).

CBFA [173] is an aspect mining approach, called Clustering-Based Fan-in Analysis (CBFA), for recommending aspect candidates in the form of method clusters, instead of single methods. The source code is first analyzed and parsed into a set of methods. Each of the method analyzed is converted into a vector based on its signature. CBFA uses a lexical based clustering approach to identify method clusters based on the similarity of vectors, and rank the clusters using a ranking metric called cluster fan-in. CBFA is evaluated on two different systems, C-based Linux, and Java system.

BMN [21] mines patterns of method calls from source code, for the purpose of code completion. The key idea of the work is, given a client method in which a number of API methods have been called on a variable, find other client methods where similar methods have been called on a variable of the same type, and recommend method calls missing within the query context, in order of popularity. BMN works on object-oriented source code that produce item sets for variable contexts. The variable context aggregates all methods called on an object-type variable within a client method. BMN uses k-Nearest Neighbor (kNN) classification. The basic idea of kNN is to find the code snippets most similar to the context for which recommendations are desired, and to generate recommendations based on the item-sets found in these snippets. BMN evaluation involves a systematic assessment of four different recommendation algorithms for autocompletion using a cross-validation design on data for client of the SWT toolkit. Specifically, the evaluation compares the recall and precision of recommendations produced with the default Eclipse algorithm (alphabetical), the frequency algorithm (most popular), association rule mining, and their own kNN-inspired algorithm. However, both the kNN-inspired and association rule techniques are shown as much superior to either frequency or alphabetical-based recommendations.

Acharya *et al.* [**3**] present a misuse detector for C. Their approach distinguishes normal paths, i.e., execution paths from the beginning of the `main` function to its end, from error paths, i.e., paths from the beginning of the `main` function to an `exit` or `return` statement in an error-handling block. It uses push-down model checking to generate such paths as sequences of method calls and applies frequent-subsequence mining to find patterns. Acharya *et al.* find project-specific patterns of method calls and conditions.

MAPO [**178**] is an API usage mining framework based on sequential pattern mining [7] for recommending code samples. A mined pattern describes that in a certain usage scenario, some API methods are frequently called together and their usage follow some sequential rules. For mining specifications, MAPO considers extracted call sequences as their observations, and mine frequently called sequences of API methods. MAPO was applied on 20 open source projects ($141K$ lines of code in total, which use Eclipse Graphical Editing Framework (GEF)), and acquired 93 patterns, which include 157 API method call sequences and cover the usages of 856 API methods.

Alattin [**149**] is a misuse detector for Java, specialized in alternative patterns for condition checks. For each target method `m`, it queries the code-search engine Google Code Search to find example usages. From each example, it extracts a set of rules about pre- and post-condition checks on the receiver, the arguments, and the return value of `m`, e.g., "`boolean` check on `return` of `Iterator.hasNext` before `Iterator.next`" or "`const` check on `return` of `ArrayList.size` before `Iterator.next`." It then applies frequent item-set mining on the sets of these rules to obtain frequent patterns. For each such pattern, it extracts the rule sets that do not adhere to the pattern and repeats mining on these, to obtain infrequent patterns. Finally, it combines all frequent and infrequent patterns for `m` by disjunction. Alattin mines sequential patterns that consist of an ordered pair of API elements (a, b), indicating that the usage of element a should occur before b in a program's execution. Alattin learns cross-project patterns of method calls and conditions.

CAR-Miner [**150**] is a misuse detector for C++ and Java. For each analyzed method `m` in a given code corpus, it queries the code-search engine Google Code Search to find example usages. From the examples, it builds an *Exception Flow Graph* (EFG), i.e., a control-flow graph with additional edges for exceptional flow to and within `catch` and `finally` blocks. From the EFG, it generates *normal* call sequences that lead to the currently analyzed call and *exception* call sequences that lead from the call along exceptional edges. Subsequently, it mines association rules between normal sequences and exception sequences. CAR-Miner detects cross-project patterns of method calls and exception-handling.

GrouMiner [110] is a misuse detector for Java. It creates a graph-based object-usage representation (GROUM) for each target method. A GROUM is a directed acyclic graph whose nodes represent method calls, branchings, and loops and whose edges encode control and data flows. GROUM associates events in a directed acyclic graph (DAG). This graph can handle special nodes to represent control structures, such as loops and conditions. Furthermore, edges not only represent sequencing constraints, but also data dependencies. GrouMiner uses an a-priori-based algorithm to detect frequent-subgraphs [128] on sets of such GROUMs, to detect recurring usage patterns. A-priori-based algorithms start from frequent single-node subgraphs and recursively extend known, frequent subgraphs by frequently adjacent neighbor nodes. GrouMiner finds project-specific patterns.

DMMC [91] learns patterns of method calls for detecting missing method calls in source code. DMMC collects statistics about *type-usages*. A type-usage is simply the list of methods called on a variable of a given type in a given client method. DMMC then uses this information to detect other client methods that may need to call the missing method. DMMC works on object-oriented source code that produce item sets for variable contexts. The variable context aggregates all methods called on an object-type variable within a client method. For a given variable x of type T, DMMC generates the entire collection of usages of type T in a given code corpus. From this collection, it computes various metrics of similarity and dissimilarity between a type usage and the rest of the collection. DMMC uses a statistical approach. DMMC detects missing method calls in SWT clients. Inspection of the results provides the additional insight that although the approach can recommend method calls with excellent performance, it is much less obvious to know how exactly to use the recommended method in that scenario: what arguments to pass in, what to do with the return value, etc.

OCD [40] is a misuse detector for Java. To mine and check temporal patterns, OCD observes a sliding window-technique that considers a limited sequence of events from the method-call traces and identifies pairs of subsequent calls to the same receiver based on finite state machines. If no second call occurs within the window, it considers the first call as isolated. OCD mines sequential patterns that consist of an ordered pair of API elements (a, b), indicating that the usage of element a should occur before b in a program's execution. Both types of occurrences serve as evidence (or counter-evidence) for temporal patterns. OCD uses multiple thresholds to decide—based on the collected evidence and counter-evidence—whether a pattern should be enforced. OCD finds project-specific patterns of method-calls and iterators.

TAUTOKO [34] uses automatic generation of test cases for mining specifications based on execution traces. TAUTOKO generates test cases to cover all possible transitions between all observed states. These transitions can either end in legal states, thus indicating additional legal interactions, or they can raise an exception, thus indicating illegal interaction. TAUTOKO makes use of finite state automaton to describe the transitions between object states. Automaton states represent different states of an object, and transitions are labelled with method names. TAUTOKO is evaluated on a sample of 800 defects seeded into six Java subjects.

Gruska *et al.* [44] present a lightweight, language-independent parser that is able to perform analysis of programs written in languages such as C, C++, Java, PHP, and others with a similar syntax for analyzing source code statically. Gruska *et al.* leverage the JADET approach to detect object usage anomalies, extending it to arbitrary languages with function calls. Their approach mines sequential patterns that consist of an ordered pair of API elements (a, b), indicating that the usage of element a should occur before b in a program's execution.

SpecCheck [101] is a misuse detector for Java. It uses the LM miner [38] to obtain specifications of the form `consequences` $\hookrightarrow$ `premises` from method-call traces, where both the premises and consequences are sets of method calls. To determine the subset of significant specifications, SPECCHECK removes the premise method calls from the instances of the specification in the training codebase and executes the mutated program. If this causes an exception at a consequence method call of a specification, for at least one instance of the specification, SPECCHECK consider this specification as significant. Otherwise, it drops the specification. For specification with multiple premise method calls, SPECCHECK repeats this check with a leave-one-premise-out strategy and drops premises whose omission does not indicate significance of the specification. Finally, SPECCHECK combines multiple specification with the same consequences by conjunction. SPECCHECK finds in this way project-specific patterns of method calls with call order.

Tikanga [157] is a misuse detector for Java that builds on the same algorithm as JADET. It replaces JADET's simple call-order properties by general Computation Tree Logic (CTL) formulae on object usages. TIKANGA combines static analysis with model checking to mine CTL formulas. Specifically, it uses formulae that require a certain call to occur in a usage, formulae that require two calls in a certain order, and formulae that require a certain call to happen after another. It uses model checking to determine the subset of all those formulae. It then applies Formal Concept Analysis [41] to obtain patterns. The learned patterns are sequences of method/function calls. TIKANGA's capabilities are the same as JADET's. The evaluation by the authors of TIKANGA applied the detector to six projects individually, finding project-specific patterns.

Lee *et al.* [64] propose 56 micro interaction metrics (MIMs) that leverage developers' interaction information stored in the Mylyn data. Mylyn is an Eclipse plug-in, which captures developers' interactions such as file editing and selection events with time spent. First, files are collected as instances, and then post-defects are counted for each file. For the regression model, the defect numbers are predicted. For classification, a file is labelled as *buggy* if it has any post-defect (post-defect number >= 1), or *clean* otherwise. Finally, prediction models are trained using machine learning algorithms implemented in Weka [49]. The trained prediction models classify instances as *buggy* or *clean* (classification), and predict the post-defect numbers (regression). The build models are file-level defect predictors. Since Mylyn contains interactions, the approach identifies common patterns of sequential events.

Pradel *et al.* [121] present a misuse detector for Java, specialized on multi-object method-call protocols. Is uses a dynamic specification miner [117, 118] to obtain multi-object specifications from method-call traces. The resulting specifications are finite-state automata (FSA), where transitions are method calls and states represent the respective objects' state. The mined FSA specifications are transformed into Fusion specifications [58], i.e., into triples of a set of relationships between the involved objects, a set of call preconditions, and a set of call effects on relationships and object states. Fusion performs a static inter-procedural analysis to verify the specifications on a given target codebase. Their approach learns patterns of method calls and conditions.

Precise [172] is an automated technique in recommending API parameters. The basic idea of Precise is to extract usage instances from existing programs and adaptively recommend parameters based on these instances and the current context. Precise mines existing code bases, uses an abstract usage instance representation for each API usage example, and then builds a parameter usage database. Upon a request, Precise uses k-nearest neighbor (k-NN) queries on the usage database for abstract usage instances in similar contexts and generates parameter candidates by concretizing the instances adaptively. Precise ranks its recommendations with respect to the similarity of context and the frequency of usage, helping developers select the right parameters more easily. Precise implementation is combined with Eclipse JDT.

NGSE [51] is a corpus based n-gram model suggestion engine for Java. The NGSE uses a trigram model built from a project corpus. After each token, NGSE uses the previous two tokens, already entered into the text buffer, and attempts to guess the next token. currently based on a static corpus of source code. The language model estimates the probability of a specific choice of next token; this probability can rank the order of the likely next tokens.

Buse *et al.* [24] present an automatic technique for mining and synthesizing succinct and representative human-readable documentation of interfaces from the Java SDK. The proposed algorithm is based on a combination of path sensitive data-flow analysis, clustering, and pattern abstraction. The approach models API uses as graphs describing method call sequences, annotated with control flow information. Concrete uses are then abstracted into high-level examples. Because a single data-type may have multiple common use scenarios, clustering is used to discover and coalesce related usage patterns before expressing them as documentation. It produces output in the form of well-typed program snippets which document initialization, method calls, assignments, looping, constructs, and exception handling.

UP-Miner [154] mines succinct and high-coverage usage patterns of API methods from source code. UP-Miner includes an API parser to parse the source code files. The API parser constructs AST trees for each source code file. After identifying API methods from their call sites in the source code files, API method sequences are collected. UP-Miner is based on the BIDE [155] algorithm to mine frequent closed API-method invocation sequences and include a two-step clustering strategy before and after BIDE to identify usage patterns. UP-Miner mines the API usage patterns and presents the resulting patterns as probabilistic graphs, which are ranked by the number of occurrences.

Given a user-specified API method, UP-Miner can automatically search for all usage patterns of an API method and return associated code snippets as reuse candidates. The approach is evaluated on a Microsoft code-base.

SLANG [129] is a code completion engine based on statistical language models. Given a Java program with holes, SLANG synthesizes completion for holes with the most likely sequences of method calls. They use a simple and scalable static analysis that extracts sequences of method calls from a large codebase, and index these into a statistical language model. Then, they employ a language model to find the highest ranked sentences, and use them to synthesize a code completion. Their approach is able to synthesize sequences of calls across multiple objects together with their arguments. However, the developer is expected to know the positions in the code where missing method calls need to be inserted when synthsizing SLANG.

CodingTracker [100] present an approach that identifies frequent code change patterns from a fine-grained sequence of code changes. CodingTracker records the code changes as soon as they are produced by developers. Consequently, the approach is the most fine-grained representation of code evolution. CodingTracker records the detailed code evolution data ranging from individual code edits up to the high-level events like automated refactoring invocations. The collected raw data is then transformed into code changes as *add*, *delete* and *update* operations on the underlying AST. Next, distinct kind of code changes are represented as combinations of the operation and the type of the affected AST node. The instances of code change kinds serve as input to a frequent item-set pattern mining based algorithm. For each mined code change pattern, the algorithm reports all occurrences of the pattern in the input sequence of code changes. For evaluating the approach, CodingTracker was installed as an Eclipse IDE plug-in.

JSMiner [107] is a static graph-based mining approach for inter-procedural, data-oriented JavaScript (JS) usage patterns. It is based on JSModel, which is a graph-based representation of JS usages. JSModel contains function calls, field accesses, control nodes, (un)named data nodes for variables, object literals, JS functions, and HTML elements. The structure of an object is captured via (un)named data nodes and edges that represent their containment relations. Edges are also used to model control and data flow dependencies among nodes. The usefulness of the approach is evaluated in two applications: detecting anti-patterns (buggy patterns) and documenting JS APIs via pattern skeletons.

PBN [123] is an intelligent code completion system for recommending the next method call, based on Bayesian networks. PBN learns typical usage pattens of frameworks from data, which are extracted by statically analyzing Java source-code repositories. On the extracted usages, PBN applies clustering techniques to improve model sizes of the learned patterns. The learned patterns are passed to the Bayesian network, which uses context information to calculate the recommendations.

MLUP [137] is a technique for mining Multi-Level API Usage Patterns to exhibit the co-usage relationships between methods of the API of interest across interfering usage scenarios. A multi-level API usage pattern is defined as a group (cluster) of API methods, that are frequently and uniformly used together across variable client programs, and regardless of the usage context. The rational behind the multi-level distribution of methods in a usage pattern is to identify the pattern's core, which represent the pattern's methods that are 'always' used together, and to reflect inferring usage scenarios of the pattern's core and the rest of the API methods. Hence, multi-level usage patterns add a new dimension that can be used to enhance the API documentation with co-usage relationships between methods of the API of interest. MLUP takes as input the source code of the API of interest and multiple client programs making use of this API. First, the API's and client programs source code is statically analyzed to extract the references between the methods of the client programs and the public methods of the API. Second, a usage vector is computed for each API public method, which encodes information about its client methods. Finally, cluster analysis are applied to group the API methods that are most frequently co-used together by client methods. MLUP is evaluated on four different APIs: HttpClient[5], Java Security[6], Swing[7], and Awt[8].

Wang *et al.* [156] present an automatic approach to identify the high level action implemented by Java loops. Loops are characterized as feature vectors in terms of certain data flow, structural, and linguistic features learned from a large corpus of open source code. The source code representation of the loop is analyzed to extract its representative feature vector. This enables clustering of various loop structures that perform the same action, identifying in this way the high level actions implemented by loops. The approach can automatically insert internal comments and provide additional higher level naming for loop actions.

White *et al.* [164] combine deep learning with software language modeling in order to improve the quality of the underlying abstractions, by providing new ways to mine and analyze sequential data, e.g., streaming software tokens. They apply deep learning models on Java projects source code files based on lexically analyzed source code written in any programming language, and other types of artifacts. White *et al.* experiment with two of the models' hyper-parameters, which govern the capacity and the amount of context used to inform predictions.

DroidAssist [111] is a misuse detector for Dalvik Bytecode (Android Java). It generates method-call sequences from source code and learns a Hidden Markov Model from them, using a modified version of the Baum-Welch algorithm, to compute the likelihood of a particular call sequence. DroidAssist analyzes a given method sequence in existing code report. If it is a suspicious API usage (i.e. is rarely used or unlikely to be used), DroidAssist can offer fixes with more probable method sequences based on their likelihoods of appearance in the existing code context.

DeepAPI [46] is the first approach to adapt deep learning to generate API method call sequences. DeepAPI formulates the API learning problem as a machine translation problem: given a natural language query, translate it into an API sequence. To annotate the API sequences with natural language descriptions, DeepAPI extracts method-level code summaries, specifically, the first sentence of a documentation comment for a method. For each method, DeepAPI traverses its AST and extracts the JavaDoc

[5] http://hc.apache.org/httpclient-3.x/

[6] https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html

[7] https://docs.oracle.com/javase/7/docs/technotes/guides/swing/

[8] https://docs.oracle.com/javase/7/docs/api/

comment part. DeepAPI learns the sequence of words in a query and the sequence of associated APIs. DeepAPI adapts an neural language model named RNN Encode-Decoder by considering the importance of individual APIs. DeepAPI is evaluated on more than 7 million annotated Java code snippets collected from GitHub. However, the authors accept that not all methods come with JavaDoc comments, and some of methods contain even irregular comment annotations.

ApiRec [102] is based on statistical learning from fine-grained code changes and from the context in which those changes were made, to recommend the next method call. Grouping fine-grained changes by high-level intent allows ApiRec to cut through the noise of unrelated tokens that may surround the recommendation point. The changes that belong to higher-level intents will co-occur more frequently than non-related changes. ApiRec works in three steps: (1) It builds a corpus of fine-grained code changes from a training set, by iterating over commits and detecting the differences in Abstract Syntax Trees (AST) nodes. (2) It statistically learns which fine-grained changes co-occur in the same changed file. Additionally, the model operates on the code context of fine-grained changes (e.g., preceding method calls) (3) It computes and then recommends a new API call at a given location based on the current context and previous changes. If it determines that an API method is indeed likely, then it returns a list of candidate API calls ranked by the computed likelihood of being selected by a developer. ApiRec is trained and tested on randomly selected Java projects from GitHub that have long development histories.

HAPI [112] is an automatically statistical approach to learn patterns of method calls from byte code of Android mobile apps. HAPI is trained based on method calls sequences extracted from ARUS, a graph-based representation of API usage scenarios, for the purpose of method call recommendations in code completion engines.

Salento [96] is a misuse detector for Dalvik Bytecode (Android Java). It uses symbolic execution to identify objects of the APIs' types and encodes each respective usage as a bag of all method calls on such an object. For each method call, Salento also encodes boolean predicates that capture constraints on call parameters, whether the call throws an exception, or other properties. Salento assumes that each usage $\texttt{U}$ conforms to a specification $\texttt{Z}$, which is unknown. It further assumes that the method calls $\texttt{X}_{\texttt{U}}$ appearing in $\texttt{U}$ inform about $\texttt{Z}$. The uncertainty about $\texttt{Z}$ is formalized as $P(Z\|X = \texttt{X}_{\texttt{U}})$, where Z is a random variable over specifications and X is a random variable over method calls in usages. Moreover, Salento expresses the uncertainty regarding the behaviors $\texttt{Y}$ of $\texttt{U}$ as $P_{\texttt{U}}(Y)$, where Y is a random variable over behaviors, and allows for a distribution $P(Y\|Z = \texttt{Z})$ over the behaviors of usages that implement a given specification $\texttt{Z}$. In this framework, Salento learns a joint distribution $P(X, Y, Z)$ from the usage examples in the training code. Salento detects patterns of method calls.

Liu *et al.* [70] collect and track a large number of fixed and unfixed violations across revisions of software, and conduct empirical analyses on identified violations and fixed violations to investigate their recurrences, their code patterns, etc. They apply heuristics in order to represent code as a set of abstract and concrete entities. The approach encodes a fixing change into a vector space using Word2Vec [42], extracts discriminating features using Convolution Neural Networks (CNNs) and regroups similar changes into a cluster using *X-means* clustering algorithm. The identified patterns are then applied to unfixed violations. The authors investigate violations and violation fixing changes collected from 730 open source JAVA projects.

ExampleCheck [174] is an API usage mining framework that extracts patterns from over $380K$ Java repositories on GITHUB and subsequently reports potential API usage violations in STACK OVERFLOW posts. The API usage violations reported are caused by three main reasons - missing control constructs, missing or incorrect order of API calls, and incorrect guard conditions. EXAMPLECHECK uses program slicing to remove statements that are not related to a given API, which improves accuracy in the mining process. It combines frequent subsequence mining to retain important API usage features, including the temporal ordering of related API calls, inclosing control structures, and guard conditions that protect an API call. EXAMPLECHECK targets 100 Java and Android APIs that are frequently discussed on STACK OVERFLOW.

MuDetect [13] is a misuse detector that automatically learns frequent API-usage patterns from Java projects. MUDETECT encodes API usages as API-Usage Graphs (AUGs) that captures all usage properties based on data and control flow analyses. AUG is a directed, connected multigraph with labelled nodes and edges. Node represent data entities, such as variables, and actions, such as method calls, conditions, iterators and exceptions; edges represent control and data flow between entities and actions represented by nodes. MUDETECT employs a code-semantic-aware, greedy frequent-subgraph-mining algorithm to mine the patterns.

FOCUS [108] is a recommender system for mining API function calls and usage patterns based on collaborative-filtering. FOCUS mines open-source project repositories to recommend API method invocations and usage patterns by analyzing how APIs are used in projects similar to the current project. This technique considers both project and declaration similarities to recommend API function calls and usage patterns. The code parser of FOCUS extracts method declarations or invocations from the source code or byte of the projects. FOCUS is evaluated on a large number of JAVA projects extracted from GITHUB and Maven Central.

2.4.3 Other Learning Approaches

Above we describe the most closely related work to the work presented in this thesis. However the literature include much more work in the area of knowledge discovery from source code artifacts, which we present in the following.

API specifications: To mine specifications, many approaches have been proposed that mainly rely on existing client code. However, as presented in Section 2.2.2, for specific APIs, client code might not be available or is insufficient. As a consequence, for such APIs, other approaches have been proposed that consider its documentation as the main source for automatically inferring specifications.

Doc2Spec [180] uses a NLP technique to analyze natural language API documentation to infer resource specifications. The inferred specifications are then used to detect bugs in open-source projects. On the other hand, D2SPEC [167] uses semi-structures online

API documentation (typically in form of HTML pages) for automatically extracting web API specifications. Given a seed online documentation page of an API, D2Spec first crawls all documentation pages on the API, and then uses a set of machine-learning techniques to extract the base URL, path templates and HTTP methods - from API documentation pages containing free-form text and arbitrary HTML structures. More specifically, D2Spec uses classifiers and a hierarchical clustering algorithm to extract a base URL and path templated for an API, and searches the context of a path template to infer the HTTP method.

Bug reports: Bug reports published in bug repositories are usually composed of a mixture of sentences in software language and natural language, and the domain-specific predefined fields. For this reason, many approaches are proposed for automatically generating bug reports summaries, which are an effective way to reduce considerable time in wading through numerous bug reports. BNER [181] is an approach for bug-specific entity recognition based on Conditional Random Fields (CRF) model and word embedding technique. DeepSum [65] is an unsupervised approach that integrate the bug-report characteristics into a deep neural network for generating bug report summaries.

Clone detection: Deep Learning (DL) can effectively replace manual feature engineering for the task of clone detection. Source code can be represented at different levels of abstractions: identifiers, abstract syntax trees, control flow graphs, and byte-code. Tufano *et al.* [153] applies neural networks on the different representation of source code to identify similarities (clone detection) in source code. They conjecture that each code representation can provide a different, yet orthogonal view of the same code fragment, thus, enabling a more reliable detection of similarities in code.

Code-example retrieval: Many other approaches do not synthesize different API usages found in source code into patterns, but instead present to developers previously written code snippets by searching through large-scale codebases based on a user-query. For example, MUSE [95] is an approach that mines and ranks code examples to show concrete usages for a specific API method. In this case, the user-query is represented by the API method of interest. CODEnn [45] is based on deep neural networks for suggesting relevant code snippets to developers to complete a task at hand. CODEnn jointly embeds code snippets and natural language descriptions (in the form of commented methods) into a high-dimensional vector space. In such a way, code snippets related to a natural language query can be retrieved according to their vectors.

Code synthesis: Existing heuristics methods in pairing the title of a post with the code in the accepted Stack OverFlow (SO) answers are limited both in their coverage and the correctness of the NL-code pairs obtained. Yin *et al.* [170] propose a method to mine high quality aligned data from SO using two sets of features: hand-crafted features considering the structure of the extracted snippets, and correspondence features obtained by training a probabilistic model to capture the correlation between NL and code using neural networks. These features are fed into a classifier that determines the quality of mined NL-code pairs. The method uses for training labelled examples. Reasonable results are achieved even when training the classifier on one language and testing on another (Java, Python), showing promise for scaling NL-code mining to a wide variety of programming languages beyond those for which we are able to annotate data. At the same time, Peddamail *et al.* [115] uses neural networks for the task of code summarization to automatically generate a natural language summary for a given code snippets. It is based on using a dataset of pairs $< NL, code >$ for training their model. Hu *et al.* [55] uses neural networks to automatically generate comments for method declarations.

Software artifacts classification: Software artifacts provide insights into how people build software. Ma *et al.* [76] propose an automated approach based on machine learning techniques for automatic classification of these software artifacts into open-source applications.

Test generation: Borges *et al.* [19] mine associations between `UI` elements and their interactions from the most common applications. Once mined, the resulting `UI` interaction model can be easily applied to new apps and new test generators. For example, AppFlow [54] is a system for synthesizing robust, reusable `UI` test. It leverages machine learning to automatically recognize common screens and widgets, relieving developers from writing ad hoc, fragile logic to use them in tests. It enables developers to write a library of modular tests for the main functionality of an app category. It can then quickly test a new app in the same category by synthesizing full tests from the modular ones in the library. By focusing on the main functionality, AppFlow provides ”smoke testing” requiring little manual work. Optionally, developers can customize AppFlow by adding app-specific tests for completeness.

Variable names generation: Jaffe *et al.* [57] present a machine translation approach to generate meaningful variable names for decompiled code. They consider decompiler output to be a noisy distortion of the original source code, where the original source code is transformed into the decompiler output. Using this noisy channel model, they apply standard statistical machine translation approaches to chose natural identifiers, combining a translation model trained on a parallel corpus with a language model trained on unmodified C code.

2.4.4 Discussion

Overall, we reviewed 65 approaches from the literature, 51 of which are closely related (Section 2.4.2) to the work presented in this thesis, and the other 14 (Section 2.4.3) differ slightly from the focus of this thesis, for example the application domain, or the type of artifact used for training their models.

The approaches presented in Table 2.1 target either C, C++, Java or JavaScript projects for learning their models. Only few approaches have been applied across multiple languages: (1) CarMiner analyzes C++ and Java code. (2) PRMiner analyzes C and C++

code. (3) Both CBFA [173] and Quante *et al.* [125] analyze Java and C code. (4) The approach proposed by Gruska *et al.* [44] can be applied on different programming languages, since it uses a language-independent parser.

11 approaches use exclusively dynamic analyses, 37 approaches use exclusively static analyses, and 3 other approaches use a mixture of both static and dynamic analyses: (1) Whaley *et al.* [163] use static analysis to deduce illegal call sequences, and dynamic analysis to extract models from execution runs. (2) Pradel *et al.* [121] use dynamic analysis to extract multi-object specifications from method-call execution traces, and static analysis to verify those specification on a given target codebase for the purpose of bug detection. (3) Weimer *et al.* [161] uses dynamic execution traces from mining specifications, and static analysis for detecting exceptional control-flow paths in source code. All the analyzed approaches are based on the following assumptions for learning API patterns: usages that occur frequently correspond to correct usages or, in other words, that *the majority of usages is correct.* This implies, that frequent usages identify correct specifications between code elements.

The analyzed approaches distinguish between three different goals: documentation and understanding of usage patterns ([85, 163]), detection of violations to usage patterns ([66, 91, 13]), and recommendation of API elements ([164, 21, 102]). Approaches used in the application domain of code recommendation, can be distinguished into: (1) recommendation of a single API element at a time, based mainly in learning unordered usage patterns ([21]), and (2) recommendation of complete code snippets ([178]).

Most of the approaches discussed in this thesis (34 approaches, 67%) fall into the category of sequential pattern mining. This is not surprising: although unordered patterns are useful and easy to implement, they are mostly limited to variants of frequent item set mining. The most commonly stated goal for mining sequential API patterns are bug detection (22 out of 34 approaches, or 69%). Robillard *et al.* [132] refer to approaches learning sequential-order patterns simply specification mining techniques. Initially, many sequential inference techniques were primarily developed for the general goal of documentation and program understanding. Lately, techniques increasingly focus on bug finding. However, unordered usage patterns can also be used to detect bugs. For example, if an approach determines that API methods `open` and `close` should be called within the same function, then the presence of an unmatched `open` method is evidence of a potential bug. Mining sequential patterns require more sophisticated analyses than for unordered patterns. The extension to sequential patterns introduces many new and challenging research problems, such as how to store abstraction of sequences efficiently and how to infer useful patterns given an observed sequence.

Program Analysis: Existing API-usage pattern learning approaches (see Table 2.1) extract information from source code using either static or dynamic analysis. Most of the surveyed approaches infer patterns that represent constraints on a single API element, typically a single reference type. There are some constraints, however, that span multiple types in combination. For instance, one may be interested in inferring that a socket's streams should only be used as long as the socket itself has not been closed. We find that only 5 of the reviewed approached infer "multi-object type properties". The reason for this is probably that single-object approaches are much easier to design and implement. Static multi-object approaches not only have to solve the aliasing problem for individual objects but also need to relate multiple objects with each other [17, 99]. Dynamic approaches must use expensive mappings to associate state with multiple combinations of objects [63].

Static Analysis: All approaches based on static analysis use code (snippets) as input for learning the patterns. Some require the code in a compiled format, such as Java Byte code [111], while others directly work on source code ([21, 110, 178]). They typically represent usages as sets, sequences, or graphs and mine patterns through frequent item-set/subsequence/subgraph mining, according to their usage representation. A strength of approaches based on static analysis is that they represent quite naturally different usage elements and their relations, since they encode usages as abstractions from how they appear in code. Another strength of static analysis based approaches is that they can train on code examples from various sources, such as documentation, code-search engines, or online sites, even if these are not compilable or executable. This makes it easier to obtain sufficiently many usage example for different APIs and enables cross-comparison of examples from different sources, which might help to mitigate biases. However, none of the approaches from the literature makes use of more than one source of usage examples, except of MuDetect that uses different sources for building its ground truth MuBench. Static approaches work directly on the artifacts related to the API of interest. Robillard *et al.* [132] distinguish static approaches between the type of artifacts they target. A popular strategy is to analyze source code that uses the API (*Client Code*). This source code does not necessarily need to be executable. Whaley *et al.* [163] infer finite-state specifications through static code analysis by inferring possible call sequences from program code. Another strategy, employed by JRF [179], is to derive rules by analyzing the code of the API itself, instead of client applications that use an API.

Dynamic analysis: Approaches based on dynamic analysis use execution traces to learn their models. They learn either association rules between (sets of) API elements or finite-state automata representing object states. A major limiting factor is that current dynamic approaches can only learn patterns of a predefined set of APIs. This set may be defined explicitly [120] or implicitly, e.g., as all APIs from a certain package [121] or library [40, 72, 101]. They need this, in order to decide whether a method call `m()` should itself become an event in the call trace or whether the execution of `m()` should be tracked to potentially add transitive calls to the trace. Therefore, dynamic detectors usually focus on learning patterns of widely used APIs, e.g., from the Java Class Library, and neglect less-commonly-used and project-specific APIs. Dynamic approaches work on data collected from a running program. A tool can read the trace online (while the program is executing) or offline by first recording a trace as the program runs and then reading the trace after the execution has terminated. Some techniques are not only online; they actually have to run the source code because they heavily interact with the running program (i.e., it is not sufficient to have pre-collected traces). A typical example in this category is OCD [40]. Furthermore, some of the dynamic approaches are not fully automatic and require additional input from a *human* expert [14].

Mining Algorithms: All approaches listed in Table 2.1 mainly use mining algorithms that aggregate the information extracted through static or dynamic analyses.

Clustering algorithms group similar code instances extracted through program analyses into clusters based on some similarity metrics: RASCAL [80] extracts information about the methods called within each analyzed class, and uses this information to group similar classes together based on the methods they call. BMN [21] and PBN [123] use respectively $k-$nearest neighbor and canopy clustering to group similar instances of an API type based on the API methods they instantiate. PRECISE [172] use static analysis to built a usage database that stores the information which parameters are used for an API in a specific context. PRECISE uses $k-$nearest neighbor to cluster usages based on similarity of their context information. FOCUS [108] uses collaborative-filtering for recommending snippets of method calls within a given context.

Statistical models calculate a probability distribution over a set of observations extracted from a training corpus. DMMC [91] and PBN [123] calculate the probability that an instance of an API type instantiate a given method call within a given context. SLANG [129] and DROIDASSIST [111] extract sequences of method calls through static analysis, and generate respectively a statistical language model and a Hidden Markov Model to calculate a likelihood probability of their occurrences within a given code context. SALENTO [96] learns a joint distribution $P(X, Y, Z)$ from the usage examples in the training code, where X is a random variable over method calls, Y is a random variable over behaviors, and Z is a random variable over specifications.

Finite State Automata (FSA) represents code as a set of state (e.g. the program state) and a transition function between the states (e.g. the instantiation of a method call, or control flow). Approaches based on FSA, usually are based on dynamic analyzes to extract program execution traces and summarize frequent interaction patterns as FSAs states and transitions [14, 40, 101, 121, 163, 166]. JADET [158] on the other hand, uses static analyzes to build a directed graph of finite state automata whose nodes represent method calls on a given object and whose edges represent control flow.

Frequent item-set mining learns patterns of API element that frequently co-occur in the training code corpus. Most of the approaches based on frequent item-set mining use static analysis to extract API element occurrences from source code, and based on some thresholds output frequent sets of such co-occurrences as patterns [21, 66, 71, 149]. Some approaches use *associate rule mining* to output patterns of frequent set pairs in the form of `antecendent` → `consequent` [72], or frequent pairs of method call sequences [150]. CODEWEB [85] mines pairs of method calls containing only one antecedent and one consequent. ALATTIN [149] generate association rules about conditions that must occur before or after a specific API call. JADET [158] collects sets of API temporal properties observed in client methods, e.g., $hasNext \rightarrow next, get \rightarrow set$ from object-specific intra-procedural control-flow graphs and provides those temporal properties to a frequent item set miner.

Sequence mining extracts sequences of API elements from source code using either static [126, 178] or dynamic analysis [3], and based on some pre-defined thresholds, learn patterns as frequent occurrences of such sequences.

Subgraph mining converts source code into a graph representation and based on some pre-defined thresholds, learn patterns as frequent subgraph occurrences from such graphs. GrouMiner [110] and MuDetect [13] convert each target method into a graph-based representation, where nodes represent method calls, conditions and iterations, and edges represent control and data flow. Then, they apply an a-prior-based algorithm on the generated graphs to identify frequent subgraph occurrences from such graphs in source code.

Deep learning is based on neural networks which are usually composed of multiple computational layers between the input and output layer. The neural network identifies a mathematical manipulation to turn the input into the output by calculating the probability of an output through each layer. Although training these models requires substantial regularization and their memory capacity is somewhat limited in practice. Learning the mapping between natural language (NL) and programming language, such as retrieving and generating code snippets based on NL queries and annotating code snippets using NL has been explored by lot of research works [8, 55, 56]. At the core of these works are machine learning and deep learning models, which usually demand for large datasets of $< NL, code >$ pairs for training (i.e. [168]). DeepAPI [46] adopts a supervised version of deep learning by using as input pairs of annotated API sequences with natural language description, information which is not always available. CODEnn [45] use deep learning for extracting relevant code snippets based on code search. DeepCode [115] automatically generates a natural language summary for a given code snippet. Deep learning methods are useful for high-dimensional data and are becoming widely used in many areas of software engineering. However, deep learners utilize extensive computational power and can take a long time to train - making it difficult to widely validate, repeat and improve their results. Furthermore, they are not the best solution in all domains. Menzies *et al.* [83] show that other classifiers perform as good as slower deep learning methods, and at the same time are at the range of 500 times faster than deep learning-base approaches.

Applications The learned patterns from each approach listed in Table 2.1 are evaluated on one of the three application domains: code recommendation, bug detection and API documentation generation. Most of the approaches (29 or 57%) are used for the purpose of bug detection in source code, using mainly precision [66, 149, 158], or both precision and recall [13] as evaluation metrics. Approaches that learn patterns for the application domain of code recommendation (17 or 33%) usually are evaluated using recall by removing some of the API elements from a code snippet, and checking if the tool is able to return the expected recommendation [21, 123]. In the meantime, very few approaches (8 or 16%) learn patterns for the purpose of automatic API documentation generation, which requires mainly a manual evaluation of the learned patterns [85]. In fact, two of these approaches are also used in the application domain of bug detection [163], and code recommendation [22].

Approaches used for the purpose of code recommendation, are based on static analysis of source code, since also the recommendations they produce should represent code snippets as they are written by developers in their code editors. On the other hand, approaches focused on bug detection use both static and dynamic analyses, since they are also interested to extract information about execution traces that might cause an exception or an invalid execution of the source code [3].

Depending on the specific application domain, some approaches apply domain knowledge filtering heuristics on the learned patterns, to further improve the quality of the results [85, 22].

Conclusions: Most of the learning algorithms used in current software engineering approaches (37%) are frequency-based, where the main intend is the aggregation of the artifacts collected through program analyses into some representative patterns. However, even though not widely used in API pattern learning approaches, NLP techniques and machine learning algorithms find applicability in other domains of software engineering. For example, AppFlow [54] adapts machine learning for the purpose of common screens and widgets recognition, Jaffe *et al.* [57] employs a machine translation approach for the purpose of variable names generation, in clone detection NLP and neural networks have successfully replaced manual feature engineering [153], end so on. Braiek *et al.* [20] examine the relationship between software development and modern Machine Learning (ML), and actually found that ML is in between the stages of early adoption and early majority. As a matter of fact, all the approaches presented in this review rely on program analyses to either represent source code in some predefined structure that guide the learning process, or to extract certain code artifact features. As a second step, different learning algorithms are applied on the extracted code artifacts to aggregate the collected information into some meaningful formats, relying mainly on simple frequency occurrences (FSA, subsequence, subgraphs, statistical methods), or similarity metrics (clustering). More advanced learning approaches are adapted in the recent years, such as neural networks, which rely on annotated (natural language, source code) pairs for training their models and generating new predictions. However, the code annotation is not always available and correct, which makes it practically difficult to build data corpus for training such models.

Since source code represent complex structure and semantics that should be captured during the learning process, program analysis are important in order to capture the right semantic of source code. In these circumstances, finding and applying the right learning algorithms that might require the least program analyses effort to extract such code semantics and at the same time be able to uncover latent knowledge from source code is in fact the main challenge. Fortunately, the machine learning community has developed many general-purpose algorithms that can automatically discover latent knowledge from data. Such algorithms we explore in Chapter 3 and Chapter 4, and evaluate if the latent knowledge they are actually able to discover is also relevant in software engineering applications.

3 Matrix Factorization to Improve Scalability in API Method Call Analytics

Software repositories, such as code repositories and bug-tracking repositories, have shifted from becoming archival entities to sources of valuable, actionable information that can be used to automatically guide development and maintenance activities [50]. Such repositories usually contain vast amounts of data that need to be processed and reason about in order to come up with useful recommendations to developers. Examples of such recommendations include which method to call next, which file is more likely to contain bugs, or a fix for some code the developer has written.

As a consequence, many automated software development and maintenance support tools have been developed over the years, which rely on analyzing large amount of data from code repositories (e.g. code recommenders). Code recommender systems inside of IDEs greatly aid programmers in writing code. It is common for such recommender systems to rely on pattern detection from the gathered data through clustering approaches to build the underlying models [172, 173]. These systems suggest method calls that are relevant to the current editing context, as shown by [21, 123, 130]. This is possible by comparing the editor content to the learned code patterns. The proposals are based on methods that other developers have used within a similar context and are sorted according to their relevance.

To be accurate, and thus useful, intelligent code completion systems need to mine large numbers of code repositories to increase the probability that the detected patterns are indeed relevant for developers. Furthermore, to increase the quality of the predicted method calls, more features such as contextual data have been considered. However, these extra features and the vast amount of data available in code repositories can lead to memory bloat, slowing down the query time and affecting the scalability of the system in general. Large model sizes require more main memory to be loaded and slow down querying time, limiting the usefulness of recommender systems in practice. Therefore, the trade-off between quality and efficiency is important, as a very slow recommender system would hinder development.

In this chapter, we investigate Boolean Matrix Factorization (BMF) as an alternative clustering technique to handle large amounts of data, by automatically removing noise data and finding the optimal patterns to represent the data space. BMF is a well-known machine learning approach used to represent big data sets through smaller dimensions, while at the same time removing noise [89]. BMF addresses the scalability issue mentioned above by breaking large matrices into smaller factor matrices. Matrix Factorization (MF) has already been shown to perform well for recommender systems used by Amazon and Netflix [61]. We want to bring the same benefits to recommender systems for software engineering to deal with increasing amounts of data. Specifically, we investigate if BMF can be used to improve analytics of code repositories in the context of intelligent method call completion.

To evaluate the effect of using BMF, we adapt the Mdl4Bmf algorithm developed by Miettinen et al. [89], which automatically calculates the factorization rank (number of clusters in clustering terminology) by using the Minimum Description Length (MDL) principle. To fully adapt BMF to code completion context, we implement a heuristic on top of the existing BMF algorithm: *the multiple assignment heuristic* (i.e., API usages that might get assigned to more than one pattern). More details can be found in Section 3.2.2.

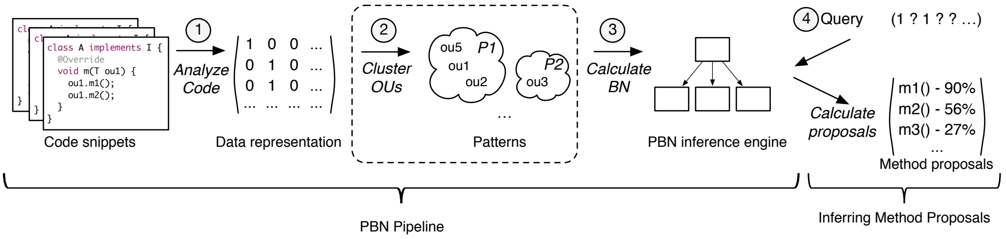
We evaluate our approach on the SWT framework APIs in the code of 3,186 plug-ins obtained from the Eclipse Kepler update site. We compare prediction quality, model size, and inference speed of BMF to those of a previous intelligent method call completion recommender that uses canopy clustering [123]. Our evaluations show that BMF greatly reduces the model size by up to 80% and improves inference speed by up to 78%, with no significant effect on prediction quality. Based on these results, we conclude that BMF is promising in the context of intelligent method call completion and speculate that other software engineering applications (i.e., artifact co-changes [171]) that rely on large amounts of input data, may also benefit from such an approach.

3.1 Background & Motivation

In this chapter of the thesis, we focus on method call completion. In other words, the developer knows the object type (s)he needs but has to decide which method has to call next. Previous work that focus on improving method call completion often used only parts of the context information available such as the type of the receiver object, the set of already performed calls on the receiver, and the inclosing method definition [21, 52, 130]. Proksch et. al. [123] showed that using additional context information such as definition sites, parameter call sites, and class context does improve the prediction quality (i.e., the F_1-measure). While introduced *Pattern-based Bayesian Networks* (PBN) and used canopy clustering to allow a better handle of the increased amount of input data, the authors found that using the additional contextual information nearly doubled the model size forcing them to consider the trade-offs between adding more useful contextual information and the increase in model size. They advocate for more intelligent machine learning algorithms that can further reduce the model size to allow using more context information.

Furthermore, pattern learning approaches often report numerous spurious patterns. Spurious patterns represent co-occurrence of instances to API elements that are found in the data but that do not correspond to sensible or useful usage patterns. The standard strategy to reduce the noise in detected usage patterns is to hand-craft filtering heuristics based on knowledge about the approach or the domain.

This part of the thesis proposes using Boolean Matrix Factorization (BMF) as a mean of building smaller models. We build BMF on top of PBN pipeline, shown in Figure 3.1,



```
class A implements I {
  @Override
  void m(T ou1) {
    ou1.m1();
    ou1.m2();
  }
}
```

Figure 3.1: Pattern-based Bayesian Network Pipeline, where clustering (dashed frame) is exchanged with BMF

where we replace the dash framed part with BMF (see Section 3.2). We start with describing the general PBN pipeline in Section 3.1.1. Next we state the problems with the current PBN recommender system in Section 3.1.2. Finally, we motivate the use of BMF by explaining how it can overcome several drawbacks of canopy clustering used in PBN in Section 3.1.3.

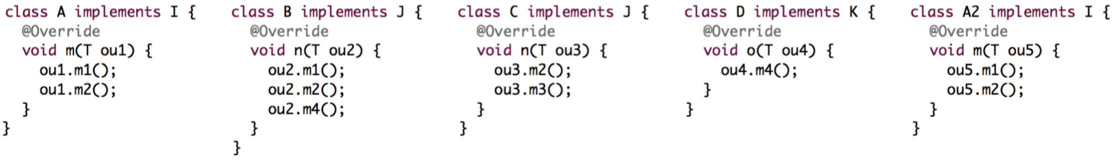
3.1.1 PBN Pipeline

The Pattern-based Bayesian Network (PBN) is an extensible inference engine for intelligent method call completion. It is structured as a four step pipeline as shown in Figure 3.1: (1) Analyze code repositories to extract *object usages* - example usages of APIs of frameworks or libraries, (2) cluster object usages to detect patterns in the data, (3) calculate the Bayesian network (inference engine) based on the detected patterns, (4) query the PBN for method proposals by providing incomplete object usages as queries. The engine returns method proposal tuples of the form *(method, probability)*, ordered by probability.

In the following, we elaborate more on each step. Step 3 (framed in Figure 3.1) is the extension point of PBN as the clustering approach can be exchanged. In our work, we replace this step by Boolean Matrix Factorization (BMF) as will be explained in Section 3.2.

Analyze Code repositories The input data for PBN is generated by statically analyzing code repositories for object usages. An *object usage* is an abstract representation of an example usage of a specific instance of an API type. It contains different *features* that describe the specific usage, such as the method calls invoked on the instance at hand, the enclosing class and the method context, the definition site, and all parameter call sites.

Figure 3.2 shows five code snippets that we use as a running example. They are analyzed to collect information for the object type `T`. The pipeline is executed separately for each object type in the API for which a recommender is built. The first object usage *ou1* has a method context `I.m` and two receiver call sites `m1` and `m2`. Thus, these three features can describe *ou1*. Note that the method context always points to the type in the hierarchy in which the method signature is defined first. Therefore it is `I.m` and not



```
class A implements I {
  @Override
  void m(T ou1) {
    ou1.m1();
    ou1.m2();
  }
}
```

```
class B implements J {
  @Override
  void n(T ou2) {
    ou2.m1();
    ou2.m2();
    ou2.m4();
  }
}
```

```
class C implements J {
  @Override
  void n(T ou3) {
    ou3.m2();
    ou3.m3();
  }
}
```

```
class D implements K {
  @Override
  void o(T ou4) {
    ou4.m4();
  }
}
```

```
class A2 implements I {
  @Override
  void m(T ou5) {
    ou5.m1();
    ou5.m2();
  }
}
```

Figure 3.2: Code snippet examples from code repositories

Table 3.1: Object Usages represented in the feature space

	in: I.m	in: J.n	in: K.o	call: m1	call: m2	call: m3	call: m4
ou1	1	0	0	1	1	0	0
ou2	0	1	0	1	1	0	1
ou3	0	1	0	0	1	1	0
ou4	0	0	1	0	0	0	1
(ou5)	1	0	0	1	1	0	0

`A.m`. This is a generalization that may lead to replications of the same object usage. For example, *ou5* of type `T` is observed in class `A2` that implements `I`. It contains the same combination of method invocations as *ou1*, and will thus have the same binary vector as *ou1*. Similar information is gathered for the other snippets.

Based on such analysis, code snippets are transformed into the processable format shown in Table 3.1, where columns represent the set of all features that appear in the code (*the feature space*). While all feature kinds (*method calls, method context, class context, definition site, and receiver call site*) are considered, the examples in this part are reduced to method calls and method context for easier illustration. Each row in the table represents a single object usage from the examples. The table may contain duplicate rows because of the context abstraction. In contrast to previous work where these are kept as separate rows in the table (such as in Table 3.1), we merge duplicates and introduce a frequency vector as will be discussed in Section 3.2.

The transformation is performed in two steps. *First,* all features are collected in a feature set that spam the available feature space in which all usages can be represented. *Second*, each object usage is transformed into a binary vector in the feature space. The dimension of each feature contained in the object usage is set to 1, all other dimensions are set to 0.

Identify Patterns The matrix generated in the previous step is passed to the clustering component in the PBN pipeline that looks for similar vectors (object usages) that can be grouped into patterns (see Figure 3.1). The clustering component is exchangeable and represents the extension point that we address in this part of the thesis. The output of the clustering step is a list of patterns. The original pipeline uses canopy clustering, which identifies three patterns from the example in Table 3.1: `P1` contains *ou1*, *ou2* and *ou5*; `P2` contains *ou3*; and `P3` contains *ou4*.

A pattern itself has a probability between 0 and 1, and every feature in the feature space also has a probability within this pattern. For example, the object usages *ou1*, *ou2* and *ou5* are similar so they end up in the same cluster. The resulting pattern *p1* has the probability 0.6, because it contains 3 out of the 5 object usages. Formally, the probability of a pattern `P` is calculated as follows:

$$p(P) = \frac{n_p}{n_{total}} \tag{3.1}$$

where n_p is the number of object usages in `P`, and n_{total} is the total number of object usages being analyzed for the given API type.

The probability of each feature in a pattern is determined by the fraction of object usages that posses this feature over the total number of object usages within the pattern being analyzed, e.g., $\frac{2}{3}$ of the usages were observed in context `I.m`. The complete vector that describes the probabilities of all dimensions in the feature space of pattern $p1$ is `(0.67,0.33,0,1,1,0,0.33)`. Formally, the probability of a feature `f` in a given pattern `P`, where n_f is the number of object usages in `P` that contain `f` is:

$$p(f|P) = \frac{n_f}{n_p} \tag{3.2}$$

To detect these patterns, PBN [123] uses a variant of canopy clustering that follows a simple algorithms:

1. Randomly select an object usage.
2. Calculate the distance to all remaining object usages.
3. Select all object usages closer than a specified threshold.
4. Merge these into a centroid that represent the cluster.
5. Remove all selected object usages.
6. Repeat steps 1-5 until no object usages are left.

The result of canopy clustering is a list of centroids together with their probabilities, calculated by relating the number of object usages that are assigned to the corresponding cluster to the total number of available object usages. In our example, three patterns (`P1`, `P2`, and `P3`) are identified: `P1` contains `ou1`, `ou2` and `ou5`; `P2` contains `ou3`; and `P3` contains `ou4`.

The output of the clustering step is a list of patterns. It is necessary that each pattern assigns probabilities to all dimensions of feature space. Additionally a pattern has a probability of occurrence itself.

Calculate Bayesian Network (BN) The patterns created in the clustering step are used to create the Bayesian network that is used to infer method proposals. Figure 3.3 shows the PBN corresponding to our running example. The root node contains all the identified patterns (three in our case) with the corresponding probabilities. The other nodes contain the different features with the corresponding probabilities within each pattern. Recall that we only show the *method context* and *method calls* here for

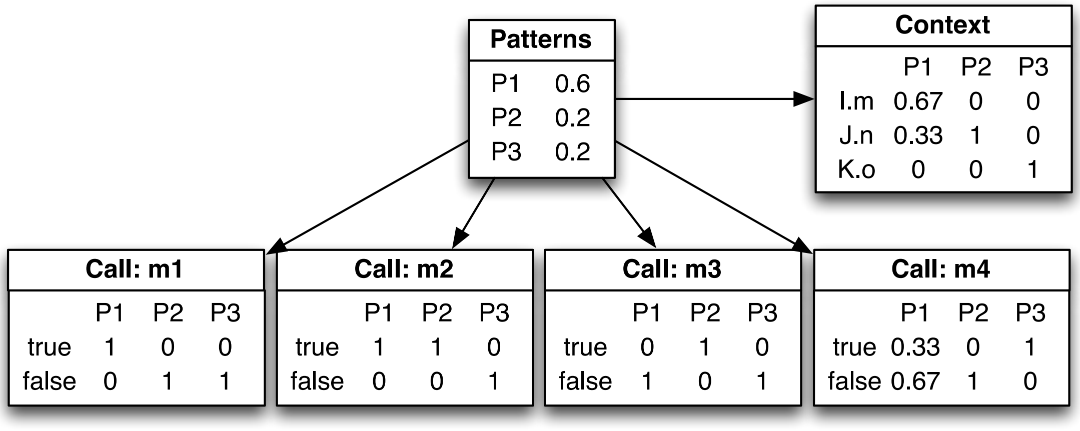


Figure 3.3: Bayesian network in the PBN inference engine

simplicity. However, the actual network would contain additional nodes for the *class context*, *definition site*, and *parameter call site*. Also note that all call nodes have two states (i.e., true and false), which denote the probability of a method to (not) be called.

Query the BN When a *query* is provided to the recommender (top right of Figure 3.1), the constructed Bayesian network is used to infer method proposals. A query is itself an object usage that was extracted from the source code under edit. All observed information is set as evidence in the network, which enables the calculation of the probability for all remaining methods (i.e., the unobserved features represented by question marks). The output of PBN is a list of method calls with an assigned probability. Only methods with a probability higher than 30% are proposed to the (hypothetical) user. While this threshold can be configured, we follow previous work [123] and select the same threshold for comparability.

3.1.2 Problem Statement

We selected the top five frameworks (according to the number of object usages) in the Eclipse plug-in dataset. Figure 3.4 shows the model sizes obtained by PBN for all API types in the selected frameworks. As the number of object usages available for a type increases, the model size linearly increases. With the increasing number of available code repositories that can be mined, a single API type can have more than 100,000 usages. A quick search for `org.eclipse.swt.widgets.Composite` (a framework-specific type) and `java.util.ArrayList` (a core Java library type) on Github returns over 220,000 and 750,000 files, respectively. If we assume each of these files contains only one object usage of the respective type and extrapolate on the shown graph, the model size for a single type would reach 75MB. A recommender system should be able to support hundreds of types. If only 100 API types would be loaded simultaneously, the model size would sum up to 7.5 GB.

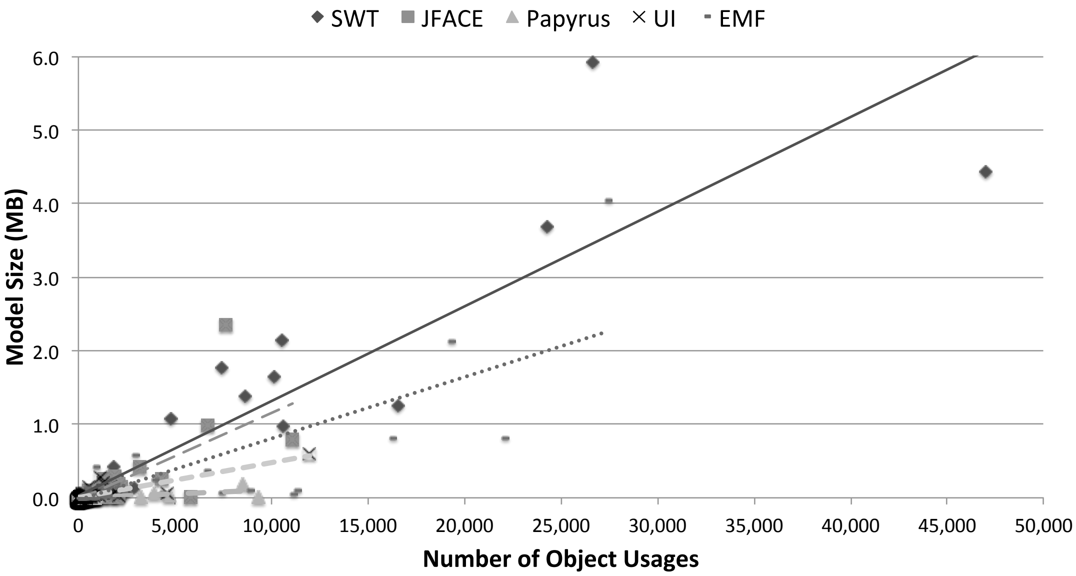


Figure 3.4: Scalability of model size in existing PBN

One problem with such large model sizes is that bloated models can greatly slow down the querying time. Additionally, models need to be loaded from hard drive and deserialized, before they can be used in the IDE. This *startup delay* can be avoided by caching loaded models. Note that a recommender may need to load multiple models in-memory to instantly support different API types that developers may use in the IDE. Smaller models consume less main memory such that it is possible to load more models at the same time, preventing unnecessary delays.

Problem 1: As more contextual information is used to describe the code in the models and as the number of input object usages increases, model sizes become increasingly big.

As more code repositories are being mined and additional context information is being used, more noise (erroneous data) is likely to appear in the models, If the clustering technique being used does not effectively filter out noise, then accurate models cannot be produced.

Problem 2: More input data may result in more noise that should be filtered to provide accurate recommendations.

Therefore, clustering has some problems that makes it less optimal. The typical clustering methods *partition* the object space, meaning that a single object cannot be assigned to multiple clusters. Further, all objects must be associated with some clusters and the algorithm cannot ignore even the most obvious outliers. Both of these restrictions are unnatural to the task, and removing them should improve the quality of the results. Another problem with clustering is the selection of the optimal number of clusters needed to represent the data. With popular clustering algorithms, such as k-means, the number of clusters must be specified a priori. The canopy clustering algorithm however, does not require setting the number of clusters, but with it, the user typically has to define two thresholds in order to determine the distance between the data points that will be clustered.

We propose to use the matrix factorization methods instead of clustering. In particular, we propose the use of the Boolean Matrix Factorization (BMF). Matrix factorization can be considered a relaxation of the canopy-clustering, where the strict partition requirement is relaxed to group similar objects together creating clusters that do not have to be disjoint or contain all of the objects. The BMF returns the possibly overlapping clusters directly, but requires all involved matrices to be binary. With BMF we can also use the Minimum Description Length (MDL) principle to automatically decide the number of clusters without any need for parameters. Hence, BMF overcomes the issues associated with clustering, and in our empirical evaluation, it typically outperforms canopy clustering.

3.1.3 Intuition Behind Using BMF

Given the above two problems, we need to find a way to reduce the size of the model without loosing important information. The pipeline in Figure 3.1 shows that the clustering technique used in Section 3.1.1 (dashed frame) affects the number of patterns detected, which in turn affects the size of the calculated Bayesian Network. Therefore, by using advanced clustering techniques, we can reduce the resulting model size. Matrix Factorization techniques provide an alternative clustering technique [28]. Previous work shows that BMF performs better with binary data compared to other MF methods [145]. Since the matrix used to represent object usages is already binary, BMF is well-suited in this context. We expect BMF to produce smaller model sizes with accurate recommendation than the currently used canopy clustering because of its following characteristics:

Identifies outliers and removes them from the data set. Canopy clustering requires that all the data points must be assigned to clusters. This means that canopy clustering cannot handle erroneous data points (*outliers*). However, big code repositories and high dimensional data usually contain lots of noise that might significantly affect the quality of the code recommenders. This issue can be resolved by BMF which is able to automatically remove noisy data during the factorization process.

Avoids user-defined parameters to cluster the data. In the background, canopy clustering uses two user-specified parameters t_1 and t_2 in order to determine the distance between the data points that will be clustered. In practice, the user has two choices. *The first* is to specify global values for t_1 and t_2 without taking into consideration that different API types would require different values (as is implemented in PBN [123], where t_1

and t_2 have very similar values). *The second* would be to perform extensive analyses for each API type individually in order to achieve better results. This of course introduces a human-involvement bottleneck and scalability issues. Miettinen *et al.* [89] propose to solve this problem by using the *Minimum Description Length* (MDL) principle [43] for Boolean Matrix factorization. By using MDL (explained in Section 3.2.1), we can automatically calculate the optimal number of patterns needed to represent every API type in the code repositories specific to the given data set.

3.2 Integrating BMF into PBN

In this section, we present our new recommender system that uses a Boolean Matrix Factorization (BMF) algorithm for generating patterns of object usages. In Section 3.2.1, we give a brief description of BMF, formalize its problem definition and show how it works. Next we describe how we integrate BMF into the code completion context in Section 3.2.2 and Section 3.2.3.

3.2.1 Boolean Matrix Factorization (BMF)

Now that we have explained the intuition behind using BMF, we discuss how BMF actually works and how we can use it to detect patterns in a given data set.

Problem Definition. Given a Boolean matrix $\boldsymbol{A}$ of size $m \times n$ and an integer k representing the expected factorization rank, find a factorization of $\boldsymbol{A}$ into a Boolean matrix $\boldsymbol{B}$ of size $m \times k$ and a Boolean matrix $\boldsymbol{C}$ of size $k \times n$ such that the error introduced by the factorization is minimized:

$$\boldsymbol{E} = min(|\boldsymbol{A} \oplus (\boldsymbol{B} \circ \boldsymbol{C})|) \tag{3.3}$$

where matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ are factor matrices of $\boldsymbol{A}$ and the pair ($\boldsymbol{B}$, $\boldsymbol{C}$) is the (approximate) Boolean factorization of $\boldsymbol{A}$. The *factorization rank* is the inner dimension of the factor matrices, in our case the $k - value$ from the previous definition. It represents the number of *clusters* in the clustering terminology. In BMF, since the data matrix $\boldsymbol{A}$, factor matrices $\boldsymbol{B}$ and $\boldsymbol{C}$, and the resulting product matrix are all Boolean, and *xor* operation (represented as $\oplus$) is used to calculate the factorization error between the original matrix $\boldsymbol{A}$ and the Boolean product of the two factor matrices $\boldsymbol{B}$ and $\boldsymbol{C}$.

The *Boolean matrix product* is defined as $(\boldsymbol{B} \circ \boldsymbol{C})(i,j) = \bigvee_k \boldsymbol{B}(i,k)\boldsymbol{C}(k,j)$; that is the normal matrix product with the addition defined as $1 + 1 = 1$. The goal of BMF is to minimize the Hamming distance between $\boldsymbol{A}$ and $\boldsymbol{B} \circ \boldsymbol{C}$, where the number of columns in $\boldsymbol{B}$ and rows in $\boldsymbol{C}$ is predefined (k). BMF requires all involved matrices to be binary. We set $\boldsymbol{A}(i,j) = 1$ if object i calls method j.

To provide the input matrix $\boldsymbol{A}$ for BMF, we use the same representation for object usages as shown in Table 3.1, which is shown again in Figure 3.5 (differences to Table 3.1 are explained in Section 3.2.2). The rows of the data matrix $\boldsymbol{A}$ represent the *object usages*, and the columns represent the *features*. Given such an input matrix, it will be

$$\begin{array}{c}\text{Frequency}\\\text{vector}\\\begin{bmatrix}2\\1\\1\\1\end{bmatrix}\end{array}\quad\begin{array}{cc} & \textbf{Matrix A}\\ & \begin{array}{ccccccc}\textit{f1}&\textit{f2}&\textit{f3}&\textit{f4}&\textit{f5}&\textit{f6}&\textit{f7}\end{array}\\ \begin{array}{c}\textit{ou1}\\\textit{ou2}\\\textit{ou3}\\\textit{ou4}\end{array} & \begin{bmatrix}1&0&0&1&1&0&0\\0&1&0&1&1&0&1\\0&1&0&0&1&1&0\\0&0&1&0&0&0&1\end{bmatrix}\end{array}\approx\begin{array}{c}\textbf{Factor matrix B}\\\begin{array}{cc}\text{P1}&\text{P2}\end{array}\\\begin{bmatrix}0&1\\1&1\\1&0\\0&0\end{bmatrix}\end{array}\circ\begin{array}{cc} & \textbf{Factor matrix C}\\\begin{array}{c}\text{P1}\\\text{P2}\end{array} & \begin{bmatrix}1&1&0&0&1&0&1\\1&0&0&1&1&0&0\end{bmatrix}\end{array}$$

Figure 3.5: An approximation of the Boolean matrix A by the Boolean product of the two factor matrices B and C. The frequency vector shows the number of occurrence for the object usages.

factored into two factor matrices, $\boldsymbol{B}$ and $\boldsymbol{C}$, shown on the right of the figure. One *factorization rank* (a column in factor matrix $\boldsymbol{B}$ and its corresponding row in factor matrix $\boldsymbol{C}$) in matrix factorization has the same meaning as a *cluster* in the clustering approaches. The object usages within a cluster are defined by the factor matrix $\boldsymbol{B}$ where $\boldsymbol{B}(i, j) = 1$ means that object usage i is in pattern j. For illustration, matrix $\boldsymbol{B}$ in Figure 3.5 indicates that *ou2* and *ou3* belong to pattern `P1`.

As in clustering approaches, similar (different) data points need to be assigned to the same (different) rank. This is ensured through matrix product algebra. Every element in row r of matrix $\boldsymbol{A}$ is equal to the sum of the products of row r in matrix $\boldsymbol{B}$ and its corresponding column in matrix $\boldsymbol{C}$.

$$\begin{array}{l}A(r,1) = \sum B(r,:) \circ C(:,1)\\A(r,2) = \sum B(r,:) \circ C(:,2)\\..\\A(r,m) = \sum B(r,:) \circ C(:,m)\end{array}$$

In this way, if there are two (or more) similar object usages in the data set, then they will have similar rows in input matrix $\boldsymbol{A}$. Following the above explanation, they will also have similar rows in factor matrix $\boldsymbol{B}$ as well. Consequently, they will be assigned to the same factorization rank in factor matrix $\boldsymbol{B}$. While the factor matrix $\boldsymbol{B}$ returns a set of patterns (clusters) to represent the object usages, factor matrix $\boldsymbol{C}$ returns the feature occurrences per pattern.

The biggest challenge with any matrix factorization or clustering approach is how to find the minimum number of patterns needed to represent a given data set. In the following paragraph, we explain how this problem is solved in the BMF context and how the factor matrices are generated. Section 3.2.2 shows how patterns are generated using BMF, and Section 3.2.3 presents the incorporation of BMF into PBN.

Generating optimal factor matrices. For generating the factor matrices, we use Mdl4Bmf, introduced by Miettinen *et al.* [90]. It is the first method that automatically selects the Boolean factorization rank for BMF without requiring any user-predefined value. MDL4BMF uses the *Minimum Description Length* (MDL) principle [43] for automatically selecting the optimal factorization rank, in combination with the ASSO algorithm [86] for generating the factor matrices B and C.

ASSO generates the factor matrices hierarchically, which means that the first column of factor matrix B and the first row of factor matrix C are generated first. It then continues to generate the second column and row, followed by the third column and row, and so forth until the user pre-defined parameter for the factorization rank is reached. In every round, it covers as much as possible from the uncovered $1s$ in the initial Boolean matrix A, and it never backtracks to previous rounds. The fact that ASSO continues this process until a user pre-defined parameter is reached, means that ASSO alone has the same problem as any other matrix factorization or clustering approaches in selecting the factorization rank (number of patterns needed to represent a given data set).

To solve this problem, Miettinen *et al.* [90] propose using the *Minimum Description Length* (MDL) principle [43] in combination with ASSO. The MDL principle selects the model that requires the minimum number of bits (model length) to encode the information in a Boolean matrix A with minimal loss. The model length for a Boolean matrix A and its approximate Boolean factorization $H = (B, C)$ is defined as:

$$L(A, H) = L(H) + L(E) \tag{3.4}$$

where E represents the factorization error (*error matrix*) shown in Equation 3.5. The total model length of the factorization, as shown in Equation 3.4, is defined as the sum of the factorization length and the error length.

$$E = A \oplus (B \circ C) \tag{3.5}$$

The MDL4BMF algorithm makes use of the hierarchical property of ASSO. In each round of the ASSO algorithm for generating a particular column and row of the factor matrices, it calculates the total model length of the respective round. According to the MDL principle, the best factorization rank is the one that minimizes Equation 3.4. In this way, when the last s steps have not improved the model length, the algorithm stops and the minimal factorization rank is returned. The parameter s represents the maximum number of larger values for factorization rank that the algorithm will try after the last decrease in model length. The user specifies the value of s. We use a value $s = 30$, which is quite a large value given the maximum number of patterns we get from our data set (around 60 patterns for the largest API type). This means that, if after 30 iterations, the model length does not decrease further, then ASSO stops and returns the minimal factorization rank with the corresponding model length.

A more detailed description of the ASSO algorithm and how the model factorization length and rank is calculated can be found in the work by Miettinen *et al.* [90].

3.2.2 Using BMF to Generate Patterns

We replace canopy-clustering with Boolean Matrix Factorization (BMF) to generate the patterns. Replacing canopy-clustering with BMF is relatively straight forward.

$$\begin{array}{c} \textbf{P1} \\ (\approx\textbf{0.4}) \end{array} \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{array}{ccccccc} \textit{f1} & \textit{f2} & \textit{f3} & \textit{f4} & \textit{f5} & \textit{f6} & \textit{f7} \end{array}$$

$$\begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.5 & 1.0 & 0.5 & 0.5 \end{bmatrix}$$

$$\begin{array}{c} \textbf{P2} \\ (\approx\textbf{0.4}) \end{array} \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 2 \end{bmatrix}$$

$$\begin{array}{ccccccc} \textit{f1} & \textit{f2} & \textit{f3} & \textit{f4} & \textit{f5} & \textit{f6} & \textit{f7} \end{array}$$

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}$$

Figure 3.6: Patterns generated from BMF

In this section, we explain how we integrate BMF into the PBN pipeline and how results might differ from canopy clustering. For this, we use the same working example from Figure 3.2. The corresponding matrix from Table 3.1 is shown on the left of Figure 3.5. The input matrix contains exactly the same input data as Table 3.1. However, instead of repeating duplicates rows (i.e., *ou1* and *ou5*), BMF introduces a *frequency vector* that stores the number of times a specific object usage is observed in the code repository. Thus, the input data matrix $\boldsymbol{A}$ is reduced to four rows, but a frequency vector is introduced that preserves count information.

Given the data matrix $\boldsymbol{A}$, BMF would produce the factor matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ shown on the right of Figure 3.5. Note that for BMF, a given object usage may be assigned to one or more patterns or to none of them (outliers). Even though *canopy clustering* is considered a *soft clustering* algorithm, the variant used in the PBN pipeline reduces it to a hard clustering algorithm because of the way distance values are configured. Each data point is assigned to exactly one cluster

To ensure that patterns closely represent their contained object usages and to be comparable with the canopy clustering configuration, we introduce a heuristic to handle corner cases where the same usage is assigned to multiple patterns. The heuristic assigns each object usage to *one* pattern that it is most similar to based on the *Hamming distance* between the feature vector of the object usage in question and the patterns it belongs to. The object usage is then removed from the other pattern(s) by changing its corresponding value to 0 in factor matrix $\boldsymbol{B}$. The object usage will only be assigned to multiple patterns if they share the smallest *Hamming distance.* For example, in Figure 3.5, *ou2* has been assigned to both patterns `P1` and `P2`. Since the *Hamming* distance to pattern `P1` is 2 and to `P2` is 3, we would remove it from `P2` and assign it only to `P1`.

3.2.3 Calculating PBN

After the patterns are detected, we need to calculate the probabilities of the patterns and of the features within a pattern. Such probabilities are calculated using Equation 3.1 and Equation 3.2, respectively. Note that the counts are taken from the introduced frequency vector.

In canopy clustering, the probabilities of all the patterns sum up to 1, but this is not the case for BMF since there are object usages assigned to multiple patterns or to none of them (outlier). This is internally handled by the BN implementation, which does a normalization of the pattern probabilities to sum up to 1.

BMF can be used in Step 2 of Figure 3.1. Its generated patterns from our example (Figure 3.6) have the same format as those generated using canopy clustering and can directly be used to calculate the BN in Step 3. After that, the inference engine can be used to infer method proposals (Step 4).

3.3 Evaluations

We use the PBN pipeline from Figure 3.1 to compare the performance of the two clustering approaches that have been discussed in this part of the thesis: canopy clustering and BMF.

3.3.1 Data

For comparability, we reuse the publicly available dataset that was previously used to evaluate PBN [123]. The dataset was obtained from the Eclipse Kepler update site, which is the main source of plug-ins for all Eclipse developers. We focus our evaluations on the `SWT` framework[1], the open source `UI` toolkit used in the Eclipse development environment. The static analyses identified 44 different API types used in our evaluation, with a total of 190,000 object usages. We use 10-fold cross-validation to evaluate each extracted API type. The object usages are disjointly assigned to 10 folds where the union of 9 folds (*training set*) is used to learn the models, and the remaining one (*validation set*) is used for querying the learned models. To avoid intra-project comparisons that may introduce a positive bias to prediction quality, we ensure that object usages generated from the same project are assigned in the same fold.

3.3.2 Recommender Evaluation

We focus our evaluation on three properties: *prediction quality*, *model size* and *inference speed.*

Prediction Quality Any new clustering approach should not have a big negative effect on the prediction quality. A big negative effect might outweigh any reduction in model size or gain in inference speed. We therefore analyze the prediction quality first. For each API type, the object usages in the validation set are used to query the model (learned from the training set). Multiple queries are constructed by randomly removing about half of the call sites from the original object usage (e.g., given an object usage with 3 calls, it is possible to create 3 queries with 1 call). We build queries that mimic both sequential (coding from up to bottom, or from bottom to up) and random (randomly adding snippets of code in the program) coding styles. The code completion engine is called on these incomplete object usages and the prediction quality is measured by calculating the F_1-measure between the ranked (list of) proposals and the removed method calls. Proposal ranking is used to filter out proposals with a probability lower than 30%. In

[1] http://www.eclipse.org/swt/

Table 3.2: F_1-measure of different recommenders

App.	PBN_{BMF}	$\text{PBN}_{\text{BMF+}}$	PBN_{15}	PBN_{40}	PBN_{60}
F_1	0.455	0.470	0.517	0.488	0.367

a last step, the different results of all queries generated from a single object usage are averaged.

Model Size We report the total model size for both approaches (canopy clustering and BMF) in Bytes. This is calculated by multiplying the number of stored float values in the Bayesian Network, representing confidence levels, by the number of Bytes needed to store float values on disk. Since each approach produces a different number of patterns, the resulting model sizes differ. Specially, more patterns result in more values to be stored in the network.

Inference Speed The inference speed measures the time needed for the code completion engine to predict the relevant method calls for a given query. Inference speed is directly related to model size. A smaller model size means that less time is needed to read the models and calculate the proposals, and vice versa. We measure this time in milliseconds and report an average inference speed for each API type (total computation time divided by the total number of queries). For each type are selected at most 3,000 queries.

3.3.3 Evaluation Results

In the following, we present the evaluation results and compare the performance of both clustering algorithms (canopy clustering and BMF) based on the three properties defined in Section 3.3.2: prediction quality, model size and inference speed.

Prediction Quality We compare PBN_{BMF} with and without the heuristic mentioned in Section 3.2.2, to the three clustered configurations of canopy clustering originally used on PBN [123]: PBN_{15}, PBN_{40} and PBN_{60}. The indices represent different distance threshold values used for canopy clustering, where smaller indices mean "stricter" clustering (more patterns).

Table 3.2 shows the F_1-measure averaged over all the analyzed API types. The table shows that our heuristic ($\text{PBN}_{\text{BMF+}}$) does have a positive impact on prediction quality compared to PBN_{BMF}. This impact is more noticeable for specific APIs.

When compared to PBN_{15}, $\text{PBN}_{\text{BMF+}}$ compromises the prediction quality (-0.047). This is expected since PBN_{15} is an almost unclustered model. $\text{PBN}_{\text{BMF+}}$ is, however, comparable to PBN_{40}. The difference (-0.018) in prediction quality is not statistically significant (p-value = 0.1257). according to the Mann-Whitney U-Test [98]. Note that $\text{PBN}_{\text{BMF+}}$ reaches a higher prediction quality than PBN_{60} (+0.103). Thus, to have a fair comparison, we only compare $\text{PBN}_{\text{BMF+}}$ with PBN_{40} in the remaining experiments.

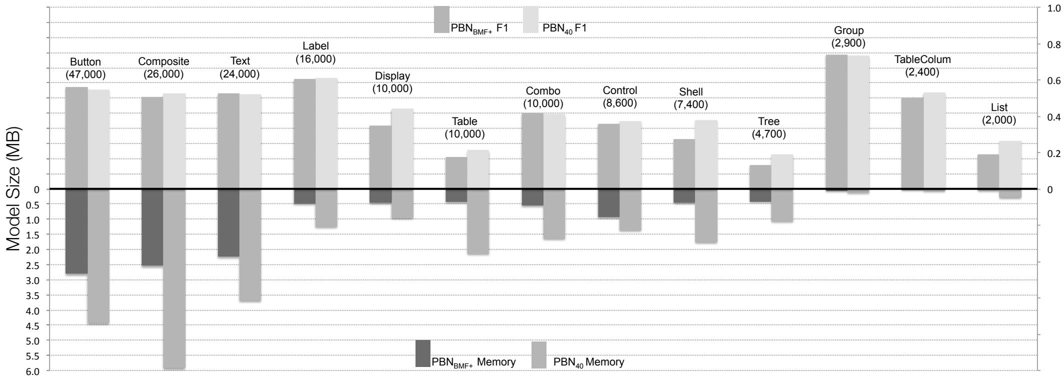


Figure 3.7: F_1-measure and model size for API types with more than 2,000 object usages. Number of object usages used for each type shown in parenthesis.

Model Size After verifying that prediction quality is not compromised, we analyze the effect of BMF on the model size. To do so, we compare the model sizes of PBN$_{\text{BMF+}}$ versus those of PBN$_{40}$. The model size depends on the number of available object usages for a type since more object usages might result in more patterns and vice versa. Therefore, we show the reduction of model size separately for each API type that has more than 2,000 object usages. We skip API types with less than 2,000 object usages since their model sizes are already small.

In addition to model size, we also show the difference in prediction quality for each of the analyzed types in order to have a fair comparison between the difference in model size and the corresponding impact on prediction quality. Figure 3.7 shows this comparison for each analyzed type, where model size is shown on the left-lower part of the y-axis and prediction quality is shown on the right-upper part of the y-axis. The plot shows that for almost all analyzed types, PBN$_{\text{BMF+}}$'s prediction quality is comparable to PBN$_{40}$, but the model sizes obtained by BMF are much smaller. This is especially obvious for types with e bigger number of object usages (more to the left), showing that PBN$_{\text{BMF+}}$ performs better for a larger number of object usages.

The reduction in model size ranges from 30% (`Button`) up to 80% (`Table`). The model size is proportional to the number of patterns created by each of the approaches. For `Table`, the model of PBN$_{40}$ contains 176 patterns on average over all folds, while the model of PBN$_{\text{BMF+}}$ contains only 36 patterns after an average of 270 outliers over all folds has been detected during factorization. In our dataset, `Table` is the type for which BMF detects the highest number of outliers. This suggests that the object usages in `Table` differ a lot. While canopy clustering creates separate patterns for these varying object usages, BMF is able to detect the ones that differ significantly from the other object usages and treat them as outliers. A closer inspection of the data shows that the object usages of `Table` are declared in very different contexts, with a rough estimation 75% of all the extracted features are related to context information. Thus, the differences between the object usages of type `Table` is related to the fact that they are declared in very different method contexts. Additionally, we see that the difference between the

F_1-measure of $\text{PBN}_{\text{BMF}+}$ and PBN_{40} for type `Table` is only 0.03. This shows that BMF is indeed removing outliers from the object usage space.

On the other hand for `Button`, BMF only detects 80 outliers averaged over all folds, even though it has almost five times more object usages compared to `Table`. However, the context information roughly accounts for only 40% of the extracted features while almost all the remaining features are definition sites. Definition sites indicate how an object becomes available in the source code but not how it is used.

For the call sites features, on average, two method calls are invoked on object usages of `Button` and `Table`. While method calls in `Button` accounts for only 0.7% of the total number of features, they account for 6% in `Table`. This suggests that even though the object usages from both types call on average the same number of methods, the total number of methods available in `Table` is almost ten times higher compared to `Button`. This means that more object usages from `Button` will be similar (have a lot of common context information and call almost the same methods), which is why fewer patterns are identified. This is true for both BMF and canopy clustering and explains why BMF results in a smaller reduction in model size here.

Inference Speed After showing that BMF reduces the model size with no significant loss in prediction quality, we expect a speed up in inference speed since a smaller Bayesian network should be faster to query. When compared to PBN_{40}, $\text{PBN}_{\text{BMF}+}$ does indeed result up to 78% faster in inference speed (from 3 *ms* to 0.6 *ms*), and 46% faster (from 3.2 *ms* to 1.7 *ms*) when averaged across all API types.

Limitations Even though the reported numbers in terms of model size (6 MB) and inference speed (3.2 *ms*) are not a scalability issue for current recommender systems, we use the same dataset as in previous work [123] for comparability. Note that this work is the first step in using BMF as a means to create smaller models within the context of intelligent method call completion, and our results show that BMF is a promising approach in this direction.

3.4 Threats to Validity

3.4.1 Internal Validity

Mdl4Bmf is a general machine learning algorithm, not bound to a specific application domain. We experimented with different values of the factorization rank for various API types to ensure that the factorization rank calculated by the algorithm is indeed optimal in our intelligent method call completion context. Our results showed that the factorization rank automatically calculated by Mdl4Bmf does indeed provide the best tradeoff between model size and prediction quality. This gives us confidence that the algorithm correctly calculates the optimal factorization rank and suggests that it is independent from the nature of input data.

3.4.2 External Validity

We test the use of BMF for one dataset within the context of one method call recommender. Different datasets and recommenders might exhibit different behaviors in terms of model size and prediction quality. Additionally, different heuristics might be required for different datasets or recommenders. For example, we analyzed the effects of the following heuristic to ensure that object usages are assigned to patterns they are most similar with: *Object usages are assigned to patterns they are most similar with based on the Hamming distance between the feature vector of the object usage in question and the patterns it belongs to.* In other datasets, the effect of this heuristic might be different. We do not generalize our results to other datasets but only point out potential applications in related work. Our work here is a first step to illustrate the use of BMF, and the PBN pipeline allowed us a fair comparison since we have all implementation details.

3.5 Related Work

Since our main goal is to address scalability and not to propose a new recommender, we do not focus on other intelligent method call completion techniques. Instead, we discuss the following four categories of related work.

3.5.1 Matrix Factorization

Matrix factorization (MF) methods are currently used in data mining to separate noise from global structure in the data [90]. They find applications in many domains such as relation extraction [28], text mining [114], recommender systems [32], data compression [90], computer vision [47], and computer networks [136]. In general, matrix factorization methods are used to represent big data through smaller dimensions while minimizing the information loss, such that the data can be reconstructed with minimal error.

The BMF approach has been recently introduced to data mining as a generalization of frequent item-set mining and database tiling [88]. Previous work shows that BMF performs better with binary data in comparison to other MF methods [145]. BMF has the requirement that all the involved matrices need to be binary (i.e. contain Boolean values). This holds for both, the initial data matrix that we want to factorize A, as well as for the resulting factor matrices B and C.

Non-Negative Matrix Factorization (NMF) [15] represents a non-negative data matrix using two factor matrices, given a pre-defined factorization rank. In comparison to BMF, NMF requires that the input data matrix and the factor matrices have non-negative values. The algorithm is shown to scale well for large data ranges [68] and is widely used in text mining and data-clustering [165]. One drawback is that we cannot use the MDL principle with NMF to calculate the optimal factorization rank for a given data set. Instead, we need to input the factorization rank as a parameter.

Furthermore, NMF creates a new problem: the rounding of the factor matrices. To obtain the final candidate clusters, we round the factor matrix $\boldsymbol{B}$ to be binary and interpret it so that if $\boldsymbol{B}(i, j) = 1$, then object i is assigned to cluster (pattern) j.

We tried NMF with our data by using the same factorization rank calculated by the BMF approach and the results were not significantly better compared to BMF. For a broader overview on different Matrix Factorization methods and their computational complexity, we refer the reader to Miettinen's work [87].

3.5.2 Potential Applications in Code Recommenders

Precise [172] is an approach to recommend parameters for method calls, and the work by Zhang et. al [173] recommends combination of method calls. They both use binary representation of the data and clustering algorithms respectively for parameter and method recommendations respectively. Since the data is already represented in Boolean format, their work might potentially benefit from BMF to construct clusters without requiring a user-specified threshold needed by their approaches.

3.5.3 Potential Applications in Pattern Mining

Frequent item-set mining is a common technique for detecting patterns in datasets. DynaMine [71], PR-Miner [66] and the work by Michail et. al. [85] are few examples in this direction. Some of these approaches [71, 85] make use of the *Apriori* algorithm to detect frequent item sets in the data. whose runtime is exponential with respect to the number of items. It is worth investigating whether it is possible to mine patterns using BMF instead. This requires the data to be represented as Boolean matrices in the form *methods* (or other code elements) by *items* (object, type etc.).

3.5.4 Scalability in Code Recommenders

Weimer et. al. [159] applied Maximum Margin Matrix Factorization to code recommenders. Their technique builds a single model for a complete framework, rather than a model for each API type. The authors point out the scalability issues they face due to the complexity of the optimization problem of the underlying factorization algorithm. This forced them to limit the size of the input data they provide to the algorithm.

Recommender techniques that treat code as plain text [51] or as some form of structured sentences with underlying statistical language models [129], naturally scale to large repositories. On the other hand, GraLan [105] needs a large number of trees/-graphs to capture the context information of the code under editing. Therefore, it uses two thresholds to limit the number and size of the generated trees/graphs. However, such techniques don't consider some of the structural information of the code, which is important to make more accurate code predictions.

3.6 Discussion

Intelligent code completion systems learn models by analyzing a large number of code repositories to increase the probability of detecting relevant patterns for developers.

With the vast increase of available data in such repositories, scalability becomes an issue especially with respect to the learned model sizes. Another factor that influences model sizes is the use of additional contextual information to improve prediction quality. In this part of the thesis, we investigate *Boolean Matrix Factorization* (BMF) as a means to create smaller models by adapting a previously developed Pattern-based Bayesian Network (PBN) framework [123], and replacing the originally used canopy clustering with BMF. Matrix factorizations are more robust than clustering approaches as they do not require partitioning. Furthermore, using the MDL principle together with the BMF allows automatic selection of the correct number of clusters.

We compare both approaches on the SWT framework, and show that we obtain model sizes that are up to 80% smaller, which in return reduced the inference speed by up to 78%, all while not compromising prediction quality (F_1-measure). In the experimental evaluation, BMF seems to be the best all-round performer, but this quality comes with a cost in running times. As each type is dealt independently, however, the work can be trivially distributed in the cloud, alleviating the problem. It is of interest to study whether we can use the specific knowledge of the domain to improve the BMF's running time (e.g. by the sparsity of the matrices, or using the Buckshot-type approach used by SONEX [84]).

However, our experimental results suggest that BMF is promising in the context of intelligent method call completion, and we speculate that other software engineering applications should benefit from it.

4 Investigating Order Information in API Usage Patterns

Application Programming Interfaces (APIs) provide means for effective code reuse. However, for the reuse to succeed, developers need to know the API and apply it correctly, i.e., according to the *API usage pattern* intended by the creators. An API usage pattern encodes a set of API elements that are frequently used together, optionally complemented by constraints like the order in which elements must be used. Researchers have proposed several *learning approaches* that are able to find API usage patterns by analyzing code repositories. These approaches commonly analyze *API usages*, i.e., code snippets that use a given API. They mine *usage patterns*, i.e., equivalent API usages that occur frequently. The proposed approaches are then used as the basis for various applications such as API documentation generation [4, 93], automated code completions [102, 123], bug or anomaly detection [91, 158], and code search [45, 74]. However, a major challenge when learning from programs is how to represent programs in a way that facilitate effective learning. All the above mentioned approaches learn indeed different pattern representations.

Our survey presented in Section 2.4.2, shows that previous learning techniques learn three different types of pattern representation:

(1) *No-order patterns* are unordered sets of frequently used code elements (e.g., [100, 102]). Such patterns encode that calls of methods, say `a`, `b`, and `c`, frequently co-occur in code, but do not feature any information about the order of calls.

(2) *Sequential-order patterns* (e.g., [117, 129]) additionally encode facts such as that `a` has to be called before `b`, and `b` before `c`.

(3) *Partial-order patterns* (e.g., [106]) are usually represented as graphs or FSMs. these patterns encode that, for example, `a` must be called first, but how `b` or `c` are called afterwards is irrelevant.

Approaches from each of the above categories justify their choice of the respective pattern representation, apply their patterns to solve a specific recommendation task, and eventually compare their results with approaches within the same category in terms of precision and recall. However, so far, we lack systematic studies of the tradeoffs between the different types of patterns in representing source code in practice. For instance, an empirical study could explore whether the increased computation complexity required to mine partial-order patterns is justified when compared to mining unordered sets.

Furthermore, previous approaches predefine the code structure (pattern type) they want to learn a-priori, by designing specific program analysis and input formats that guide the learning process. This makes it hard to judge the impact of research that addresses API usage pattern learning. A comparison of different pattern types with regards to some predefined metrics is challenging, because each approach in the literature uses a different learning technique with configurations specific to its data set (e.g., frequency threshold), a different representation for usage examples and patterns, and might even be specifically tied to a particular programming language or input form (e.g., source code vs. byte-code vs. execution traces). Moreover, the different approaches are evaluated on different sets of target projects, such that the respective results are hardly comparable. In many cases the exact versions of the projects are not reported or became unavailable, which makes it impossible to reproduce results and to evaluate other approaches on the same project versions. Ideally, we would need a unified learning technique that can provide the three different patterns representations discussed above. Applying such a learning technique to the same data set and with the same settings would provide a fair comparison between API pattern types.

In this part of the thesis, we address this challenge and present, to the best of our knowledge, the first empirical comparison of API pattern types and investigate their effectiveness in representing API usages in the wild. The different pattern types we compare, consider constraints of different nature between method calls, and thus understanding what exactly they are able to mine in a concrete setting constitutes an interesting and relevant subject in many software engineering applications (e.g. code recommendation or misuse detection). To provide a fair setting, we use a common data set of 360 open-source Github C# repositories with over 68M lines of code [122], and adopt an established mining algorithm that can be customized to mine all three types of patterns, episode mining [1]. *Episode mining* is a well known machine learning technique used to discover partially ordered sets of *events* from a stream, called *episodes* (*patterns* in our terminology). In our setting, *events* are method declarations or invocations (cf. 4.3.2). Episode mining has already been used in several domains such as neuroscience [1], text mining [2], and positional data [48]. We can mine all three pattern types discussed above by adjusting certain parameters of the episode mining algorithm. With this experimental setup in place, we can produce sequential, partial and no-order patterns using the same mining algorithm and same data set. Our experimental setup is publicly available as a benchmark (PtBench[1]), and can be used by other researchers to perform similar empirical studies.

In this first study, we compare pattern types in terms of three metrics (defined in Section 4.4.3):

Expressiveness quantifies the richness of the language corresponding to a pattern type whose grammar rules are the mined patterns. We measure expressiveness as the number of words (i.e., derived sequences of method calls) in the language. This measure indicates how well the mined patterns abstract over the variety of concrete API usages observed in source code. Conceptually, one would expect that less structure patterns encode a richer language. The question is, though, to what extent do the differences in expressiveness between pattern types materialize in *the wild*.

Consistency quantifies the extent to which the words in the language defined by the

[1] http://www.st.informatik.tu-darmstadt.de/artifacts/patternTypes/

mined patterns are actually found in the code. This is to judge how truthful the mined API usage patterns represent actual API usage constraints implicitly encoded in source code. From a practical perspective, this metric gives us insights about the relevance of the order information encoded in sequential and partial-order patterns.

Generalizability measures whether the usages a pattern encodes are specific to a single code context or if they generalize to multiple contexts. In language terminology, this metric indicates whether the learned model is applicable across domains/projects or whether we learn domain-specific languages (models). This is important to understand the applicability of the information encoded in the learned patterns.

We also present some statistics in terms of the *pattern size* that refer to the number of events within the learned patterns, and in terms of *API types* that investigates whether the learned patterns encode interactions between method calls of the same or multiple API type. Results obtained on the empirical comparison of the three pattern types, on an existing dataset of 360 C# code repositories highlight interesting evidences in terms of expressiveness, consistency, generalizability, patterns size and number of API types.

The remainder of this chapter is organized as follows: In Section 4.1, we provide an overview over related work on different API usage representations and empirical studies based on API usages. Section 4.2 introduces some conceptual differences between different pattern type representations. The adaptation of the general episode mining algorithm within the domain context of pattern mining for software engineering is presented in Section 4.3. In Section 4.4 we continue with the evaluation setup, and Section 4.5 presents the complete PtBench pipeline. In Section 4.6, we use PtBench to empirically evaluate and compare the three different pattern types that we analyze. From our empirical evaluation, we derive a set of implications that might be considered useful to other researchers working with code patterns. The derived implications are presented in Section 4.7, followed by the discussion of threats to validity in Section 4.8. At the end, we conclude our discussion in Section 4.9.

4.1 Related Work

To summarize the state of the art and motivate the work presented in this chapter, we look into two different directions of related work. The *first* is with regards to how order information is treated in existing API usage mining techniques and representations, and the *second* are empirical studies that have investigates API usages in practice.

4.1.1 API Usage Representations

We found in Section 2.4.2 that API usage representations can be divided into three types: *no-order*, *sequential-order* and *partial-order*.

No-Order Patterns The simplest form of learning API usage patterns is to look at frequent co-occurrences of code elements, while ignoring the order these code elements occur in. *Frequent item-set mining* is a typical example in this category and variations of it have been commonly used in existing code recommender systems [85, 100, 102]. For example, Saied *et al.* [139] present a hierarchical *clustering* approach for mining multi-level API usage patterns independently from their usage context, in order to enrich the API's documentation.

Sequential-Order Patterns To take code semantics into account, many API usage representations consider order information. For example, calling the constructor of an API type must happen before calling any of its methods. The patterns mined by *sequence mining* encode strict sequential order between code elements in a pattern. Existing approaches are based on, but not limited to, using information from the API's source code [3, 158], API documentation [180], program control-flow structure [126], and program execution traces [39, 117]. Statistical models have also been used to predict the next code element (e.g. method call), given a current context (e.g., sequences of already seen method calls). Examples include n-gram language models [129] or statistical generative models [112]. Additionally after identifying sequences, some techniques rely on clustering to build pattern abstractions [24, 154, 178].

Raychev *et al.* [129] use *n-gram* language models for learning sequences of method calls across multiple objects together with their arguments. Hapi [112] is a statistical generative model for learning API usages from byte code of Android mobile apps. Approaches based on *code search* extract relevant usage examples from code repositories and other sources of information. Then, clustering algorithms are usually applied on the extracted examples to cluster similar code sequences in order to build pattern abstractions. For example UP-Miner [154] and MAPO [178] mine API usage patterns from source code using a combination of both subsequence mining and clustering approaches. ApiMiner [93] and Buse *et al.* [24] use a clustering approach based on static-slicing and path-sensitive data-flow analyses respectively.

Partial-Order Patterns To allow more flexibility in representing code semantics, *partial-order* mining techniques have been considered. In this scenario, code elements `b` and `c` must occur after code element `a`, but the order in which they occur (`b` before or after `c`) is not relevant. *Graph-based* techniques like GraLan [105], GraPacc [103], and JSMiner [107] represent source code in a *graph* to identify frequent subgraph patterns. *Automata-based* techniques or Finite State Machine (FSM) represent code as a set of states (e.g. method calls) and a transition function between the states. The framework presented by Acharya *et al.* [4] extract API usage patterns directly from client code. This framework is based on FSMs for generating execution traces along different program paths. In their terminology, partial-order expresses choices between alternative code elements. In our terminology, a partial-order pattern includes strict and/or unordered pairs of code elements.

4.1.2 Empirical Studies of API Usages

Researchers have extracted API usages through mining software repositories and studied the characteristics of these usages or used them in various applications. Usage patterns are explored in [75], from the Java Standard API with an early version of the Qualitas Corpus which contains 39 open source Java applications. A study on a larger corpus (5,000 projects) on usages of both core Java and third-party API libraries is performed in [124]. The diversity of API usages in object-oriented software is empirically analyzed in [81]. In their context, diversity is defined as the different statically observable combinations of method calls on the same project. Multiple dimensions of API usages are explored in [35], such as the scope of projects and APIs, the metrics of API usages (e.g., number of project classes extending API classes), the API's metadata, and project versus API-centric views. CoupMiner [138] combines client-based (client programs) and library-based (library-code) usage pattern mining. This approach tries to bring together the advantages of both worlds: the precision of the client-based techniques and the generalizability of library-based techniques, in the mined code patterns.

Previous work often focused on comparing one learning technique with other learning techniques within the same pattern type [4, 24, 105]. For example, Pradel *et al.* [117] present a framework for evaluating different specification miners. They use the framework to evaluate three mining approaches that learn sequences of API method calls. Our work instead, focuses on understanding the trade-offs between different code pattern types (sequential, partial and no-order patterns). The empirical study on API usages presented in [176] focuses on how different types of APIs are used. Our work is mainly concerned with API patterns instead of single usages. The work in [132] provides a more comprehensive survey on API property inference and discusses over 60 techniques developed for mining frequent API usage patterns.

Overall, existing studies focus on different aspects of API usages, but do not analyze the differences between API usage pattern types. Our work fills this gap by conducting an empirical study to investigate the impact of order information on API usage patterns mined from large repositories, and their trade-offs with respect to three metrics: expressiveness, consistency, and generalizability.

4.2 Conceptual Differences between Pattern Types

Our presentation of related work has shown that existing representations of API usage patterns treat order information differently. In order to reason about the individual advantages and disadvantages, it is necessary to first understand the conceptual differences between the three pattern types.

We use the code snippets in Figure 4.1 as a running example. The two code snippets implement the same task and have the same semantics, but slightly different syntax. In both cases, an object of type `T` is created first, and depending on a condition, `m1()` or `m2()` is invoked. Figure 4.2 shows the order in which the different code elements occur in each case. Due to the different conditions, the order information between both examples differs.

```
var var1 = new T();
if (condition) {
  var1.m1();
} else {
  var1.m2();
}
var1.m3();
```

(a) Validating the condition

```
var var2 = new T();
if (!condition) {
  var2.m2();
} else {
  var2.m1();
}
var2.m3();
```

(b) Negating the condition

Figure 4.1: Different code variants with same semantics

```
new T()
var1.m1()
var1.m2()
var1.m3()
```

(a) Events from code snippet in Figure 4.1a

```
new T()
var2.m2()
var2.m1()
var2.m3()
```

(b) Events from code snippet in Figure 4.1b

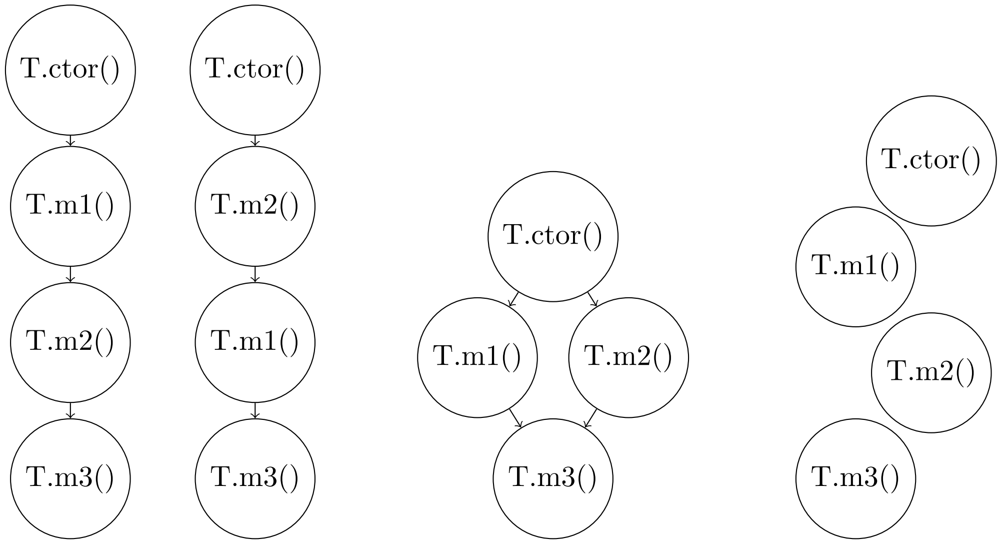
Figure 4.2: Order of code elements from Figure 4.1 extracted for pattern mining

Let's assume that the code snippet in Figure 4.1a occurs 10 times in the data set, while the one in Figure 4.1b occurs 8 times. A no-order pattern would present the methods as a set, as shown in Figure 4.3c. Two sequential-order patterns are required to represent the code examples in Figure 4.1, as shown in Figure 4.3a. On the other hand, a partial-order pattern would represent both code snippets in a single abstract pattern with an occurrence value equal to 18, as shown in Figure 4.3b. Note that there is a partial-order between `T.m1()` and `T.m2()`, because it is irrelevant which one of them comes after `T.ctor()` and before `T.m3()`.

The different pattern representations show that partial-order and no-order patterns have higher occurrence values since they combine different code sequences, compared to sequential-order patterns that represent each sequence as a separate pattern. Furthermore, partial-order patterns are able to learn important order information between method calls, information that is missed by no-order patterns. At the same time, partial-order mining avoid redundant to represent single sequences as in sequence mining.

Partial-order is especially relevant in APIs that allow to compose different elements that provide many configuration options, which do not depend on each-other, or that overload methods to provide alternatives to the developer using the API. A prominent example of such APIs are components of Graphical User Interfaces (`GUIs`) like `Swing` or `SWT`. Using `UI` components implies that the constructor is called first, but in most of the cases the order in which the elements are added to the `UI` component or how they are configured, is irrelevant. Consider, for example, the variety of elements that can be added to a `Form`, or how different applications may add these elements in completely different orders. As a result, a miner will find large amounts of code examples for `UI` s in code repositories, but with a very high variation between the examples.

Sequential-order patterns will represent each of these variations as a separate pattern, each of them with a very low occurrence value. In contrast, partial-order will represent



(a) Sequential-order (b) Partial-order (c) No-order

Figure 4.3: Pattern representations for different types

them by a single abstract pattern that defines that the `UI` element constructor should be called strictly before adding any feature to the newly created `UI` element, but that the order in which features are added is irrelevant. Obviously, a no-order pattern could manage this abstraction as well. However, the drawback is that it may over-generalize by not differentiating cases in which the order occurrence might make a difference in how an API is used. For example, `JFrame.pack()` is a method from the Java `Swing` library that is used to reduce a *frame* element to its optimal size by considering the layouts and the preferred sizes of all contained elements. Calling this method for the frame before adding all the intended `UI` elements to the frame, would cause the frame to look differently than intended.

The discussion above shows the benefits in using partial-order patterns for representing different source code variants into high-level abstract patterns, compared to sequential-order and no-order patterns. Following the same logic, partial-order patterns show an advantage compared to sequential-order patterns in cases when the training data does not contain enough API usages. Coming back to our running example, in case the mining algorithm uses a threshold frequency of 9, the sequence miner would miss learning the second sequence shown in Figure 4.1b (containing an occurrence value of 8), even though it is a valid sequence. On the other hand, the partial-order miner would combine both sequences, which increases their combined occurrence value and results in meeting this threshold.

However, there is currently no empirical evidence to support these benefits. To fill this gap, our empirical evaluations in Section 4.6 investigates whether generalizable partial-order patterns can be learned from large code repositories, and how they compare to the sequential and no-order patterns that would be learned from the same code repositories.

4.3 Episode Mining for API Patterns

We briefly overview the episode mining algorithm and then explain how we use it to mine patterns from open source C# Github repositories, in three steps: *(a)* generate an event stream by transforming source-code into a stream of events, *(b)* apply episode mining algorithm to mine API usage patterns, and *(c)* filter the resulting partial-order patterns.

4.3.1 Episode Mining Algorithm

To support the detection of sequential-order, partial-order, and no-order patterns in source code, we use the episode mining algorithm by Achar et. al. [1] for the following reasons. *First*, it facilitates the comparison of different pattern types, since it provides one configuration parameter for each type. The other option would be to use different learning algorithms, one per pattern type. In this case, ensuring the same baseline for the empirical comparisons will be difficult, since each algorithm might use different configurations and input formats. *Second*, it is a general purpose machine learning algorithm, which has performed well in other applications: text mining [2], positional data [48], multi-neuronal spike data [1]. *Third*, the implementation of the episode mining algorithm [1] is publicly available.

The term *episode* is used to describe a partially ordered set of events. *Frequent episodes* can be found in an event stream through an a-priori like algorithm [6]. Such an algorithm exploits principles of dynamic programming to combine already frequent episodes into larger ones [78]. The algorithm alternates episode candidates generation and counting phases so that infrequent episodes are discarded due to the downward closure lemma [1]. The counting phase tracks the occurrence of episodes in the event stream using Finite State Automaton (FSA). More specifically, at the k-th iteration, the algorithm generates all possible episodes with k events by self-joining frequent episodes from the previous iteration consisting of $k-1$ events each. The resulting episodes are episode candidates that need to be verified in the subsequent counting phase. A given episode is *frequent* if it occurs often enough in the event stream. A user-defined *frequency threshold* defines the minimum number of occurrences for an episode to be frequent. An *entropy threshold* determines whether there is sufficient evidence that two events occur in either order or not. All frequent episodes that fulfill the minimum *frequency* and *entropy* threshold are outputted by the algorithm in a given iteration k, and all infrequent episodes are simply discarded. The next iteration begins with generating episodes of size $k+1$. The *entropy threshold* is specific to partial-order patterns. It has a value between 0.0 and 1.0, inclusive. A value of 0.0 means that no order will be mined, resulting in no-order patterns. A value of 1.0 means a strict ordering of events, resulting in sequential-order patterns. Values between 0.0 and 1.0 result in partial-order patterns, with varying levels of strictness. We mine the three pattern types by adjusting the configuration parameter of the episode mining algorithm: *NOC* for No-Order Configuration, *SOC* for Sequential-Order Configuration, and *POC* for Partial-Order Configuration. More details about the algorithm can be found in the work by Achar et. al. [1].

4.3.2 Mining API Usage Patterns

Event Stream Generation In our context, an *event* is any method declaration or method invocation. To transform a repository of source code into the stream representation expected by the episode mining algorithm, we iterate over all source files and traverse each Abstract Syntax Tree (AST) depth-first. Whenever we encounter a method declaration or method invocation node in the AST, we emit a corresponding event to a stream. We use a fully-qualified naming scheme for methods to avoid ambiguous references. The following is how we deal with the two types of nodes we are interested in:

- Method invocation is the fundamental information that represents an API usage, for which we want to learn patterns. While a resolved AST might point to a concrete method declaration, we generalize this reference to the method that has originally introduced the signature of the referenced method, i.e., a method that was originally declared in an interface or an abstract base class. The reason is that the original declaration defines the *contract* that all derived classes should adhere to, according to *Liskov's substitution principle* [79]. Assuming that this principle is universally followed, we can reduce noise in the dataset by storing the original reference.

- Method declarations represent the start of an enclosing method context that groups the contained method calls. We emit two different kind of events for the encountered method declaration. *Super Context:* If a method overrides another one, we include a reference to the overridden method, i.e., the encountered method overrides a method in an abstract base class. This serves as context information that might be important for the meaning of a pattern. *First Context:* Following the same reasoning as for super context, we include a reference to the method that was declared in an interface that originally introduced the current method signature, which could be further up the type hierarchy of the current class.

In both cases, method declaration or invocation, the generated events have the following format: `[RT:QT] [T].M([PT] [PT] ...)`, where `RT` is the return type, `M` is the method name, `QT` is the fully qualified name of it's declaring type and `T` is it's simple type name. For constructor calls, we use the label of the form `ctor` as method name. We use the declaring type in the event signature to abstract over the different static receiver types. The `PT` label stands for parameter types, in order to distinguish overloaded methods by their parameter entities.

We apply heuristics to optimize the event stream generation. *(1)* We filter duplicated source code, e.g., projects that include the same source files in multiple solutions or that add their references through nested submodules in the version control system. *(2)* We ignore auto-generated source code (e.g., UI classes generated from XML templates), since they do not reflect human written code. *(3)* We ignore references in the data set that point to unresolved types or type elements. These cases indicate transformation errors of the original dataset, that were caused by -for example- an incomplete class path. *(4)* We do not process empty methods, nor include their method declarations in the event stream.

In addition to the heuristics mentioned above, we also ignore methods of project-specific APIs (i.e., declared within the same project) to avoid learning project-specific patterns. The reason for this is because in this part of the thesis, the goal is to learn general patterns that have the potential to be re-used across contexts. Later on, in Chapter 5 we mine patterns and detect misuses on a per project basis in order to also be able to detect misuses that come from project-specific APIs.

Learning API Usage Patterns We feed the generated event stream to the episode mining algorithm after fixing the threshold values: frequency and entropy (as evaluated in Section 4.4.2). An *episode*, outputted by the mining algorithm, represents a partially ordered set of events as a graph with labelled nodes and directed edges. Nodes represent a method declaration or a method invocation, and the directed edges represent the order in which they are called in the source code.

Figure 4.3 shows episode representations for the different pattern types. A no-order pattern would present the method calls as a set, as shown in Figure 4.3.(c). Two sequential-order patterns would be needed to present the two valid sequences presented by the partial-order pattern, respectively shown in Figure 4.3.(a) and Figure 4.3.(b). Note that methods `m2()` and `m3()` can occur in either order as defined by the partial-order pattern.

Filtering Partial Order Patterns While *SOC* and *NOC* generate episode candidates that are either sequences or sets of events respectively, *POC* might generate episode candidates from all three representations, since it contains the sequential and no-order types as special cases. In case all the episode candidates in *POC* are considered frequent episodes during the counting phase, then all of them are outputted by the algorithm. This implies that in every iteration (i.e, pattern size), *POC* might output redundant patterns containing the same set of events but differ in the order information. For illustration, assume that *POC* generates episode candidates in iteration 3 by combing the following patterns from iteration 2: $a \rightarrow b$ and $a \rightarrow c$. The episode candidates in iteration 3 will be: $a \rightarrow b \rightarrow c$ and $a \rightarrow c \rightarrow b$ as sequences, and $a \rightarrow$ *(b, c)* as partial-order, all possible orderings between the two newly connected events b and c. The partial-order episode $a \rightarrow$ *(b, c)* represents both $a \rightarrow b \rightarrow c$ and $a \rightarrow c \rightarrow b$. However, if all three episode candidates turn out to be frequent in the subsequent counting phase, the two other sequences will also be carried over to the next iteration. These redundant patterns are meaningless for source code representation though and we filter them out in each iteration.

4.4 Evaluation Setup

This chapter describes the data set we use, presents the analyses of the frequency and entropy thresholds used with the episode mining algorithm, and defines the metrics for patterns comparison.

4.4.1 Dataset

We use an established dataset that consists of a curated collection of $2,857$ C# solutions extracted from 360 GitHub repositories [122] with a total of $68M$ lines of source code covering a wide range of applications and project sizes that provide many examples for API usages. The data set uses a specialized AST-like representation of source code with fully-qualified type references and elements. This relieves us from the burden of compiling it to get resolved typing information and makes it easier to transform the source code into the event stream.[2]

We find $138K$ type declarations in the dataset that extend a base class or implement an interface. These type declarations contain $610K$ method declarations. Out of these, $50K$ (first context plus super context) override or implement a method declaration introduced in a dependency. The same dependency can be used in other projects, so focusing on these reusable methods provides valuable context information for the API usage. We find $2M$ method invocations across all method bodies of the data set.

4.4.2 Threshold Analyses

The episode mining algorithm uses two thresholds: *frequency* and *entropy*. The threshold values directly impact the number of patterns learned: higher threshold values means stronger evidence in the source code that a given pattern occurs. In this section, we empirically evaluate the effects of the threshold values on the number of patterns learned by the three configurations (NOC, SOC, POC), and select the ones to use for the empirical evaluations presented in Section 4.6.

Entropy Threshold Since this threshold is specific to POC, we first focus on analyzing the number of patterns learned by POC for different entropy and frequency thresholds. Our analyses reveal an increasing number of patterns learned for different entropy thresholds at every frequency level. This is expected, since for entropy values near to 0.0, the algorithm learns mainly unordered sets of events that abstract over several usages. On the other hand, for entropy values near to 1.0 the algorithm learns mainly sequences of events, one for each frequent sequence. For simplicity, Figure 4.4(a) shows only a few frequency levels, but similar curves are produced in other frequency levels as well. We observe that for every examined frequency level, POC learns a fairly stable number of patterns in the entropy segment of $[0.55, 0.75]$. A stable number of patterns for different threshold values, means that the patterns are not much affected by small fluctuations of the threshold values, making them more preferable compared to an unstable set of patterns that are easily affected by small changes in the threshold values. Our data analyses within this segment reveals that the minimal variation in number of patterns occur for values of $0.71 - 0.72$. Hence, we use the entropy threshold of 0.72 in our next analysis for the frequency threshold and in our empirical evaluations in Section 4.6.

[2] We use the visitors in the dataset for the transformation.

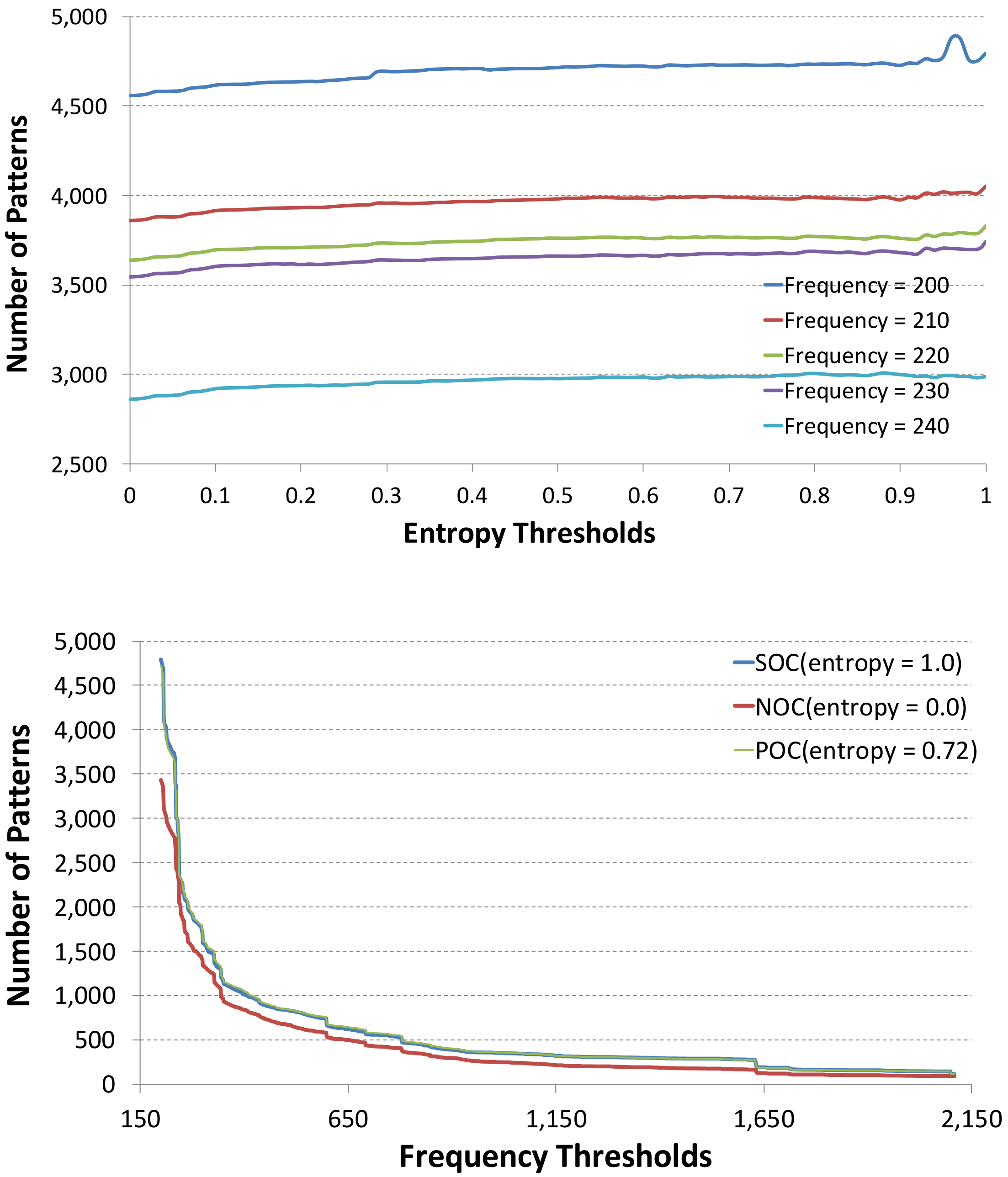


Figure 4.4: Frequency and entropy threshold analyses

Frequency Threshold Our analyses in Figure 4.4(b) show that *SOC* and *POC* learn comparable number of patterns for different frequency values, while *NOC* learns less patterns in every frequency level compared to the two others. This is due to the order information: while *SOC* and *POC* may learn multiple patterns for the same set of events, *NOC* simplifies to a single pattern. We select a frequency value that gives a good trade-off between the total number of patterns learned per configuration and comparable number of patterns learned across configurations. Our analyses reveal that this is achieved at the frequency threshold of 345, which we use in the rest of our evaluations. A comparable number of patterns across configurations avoids bias towards one configuration.

4.4.3 Metrics for Pattern Comparison

We define the following metrics to quantify different properties of the mined patterns in our experiments.

Expressiveness Using a formal language terminology, an API usage pattern can be seen as a *grammar rule* of a language over an *alphabet* of method declaration/invocation (events). The more words the sub-language it defines has, the more expressive a pattern is. A *sequential-order* pattern ($a \rightarrow b \rightarrow c$) when seen as a grammar rule defines a language with a single word $\{abc\}$. A *partial-order* pattern ($a \rightarrow (b, c)$) defines a language with two words, $\{abc,\ acb\}$. A *no-oder* pattern (a, b, c) defines a language with six words $\{abc,\ acb,\ bac,\ bca,\ cab,\ cba\}$. The expressiveness of a pattern type is determined by the number of patterns (grammar rules) it defines, and how well these patterns abstract over the variety of concrete API usages observed in source code.

To investigate how the three configurations (*SOC*, *POC*, and *NOC*) compare to each other in terms of expressiveness, we calculate three metrics for each configuration pair (`c1`, `c2`): (a) `exact(c1,c2)` is the number of patterns that are exactly the same in `c1` and `c2`; (b) `subsumed(c1,c2) = (x,y)` is a pair that represents the number of patterns x learned by `c1` that subsume y patterns learned by `c2`. We say that a pattern $p1$ subsumes a pattern $p2$ iff they relate the same set of events and all words defined by $p2$ are also defined by $p1$, e.g. the grammar rule of a no-order pattern (a, b, c) subsumes both the grammar rules ($a \rightarrow (b, c)$) and ($a \rightarrow b \rightarrow c$) from the partial and sequential-order patterns respectively; (c) `new(c1,c2)` is the number of patterns learned by `c1` that include events for which `c2` does not learn any pattern.

Consistency The three pattern types differ in the extent to which they preserve code structure. While *no-order* patterns cannot represent any structure, *sequential-order* patterns can encode an absolute order of events, and *partial-order* patterns can even represent complex control flow that is imposed by control structures like `if`. We establish the *consistency* metric as a way to quantify how important the order information encoded by sequential-order and partial-order patterns is in practice. The metric takes values in $]0.0, 1.0]$, and for a given pattern p is defined as:

$$consistency(p) = \frac{Occs(p)}{OccsSet(p)} \tag{4.1}$$

where $Occs(p)$ is the number of occurrences of p, and $OccsSet(p)$ is the number of co-occurrence of events in p regardless of their order. A high consistency emphasizes the importance of the encoded order. A low consistency means that in most cases, the respective code elements occur in an order different to the one encoded in the pattern, suggesting that the structural information encoded by the pattern is irrelevant.

Generalizability Finding instances of a pattern in multiple contexts indicates that the pattern represents an abstraction over a set of similar API usages, e.g., used by multiple developers. On the other hand, a very *local* pattern might suggest that it does not generalize beyond a specific context, e.g., it might only be used by a specific developer. To quantify the *generalizability* of a pattern, we count the number of contexts in which we can observe it at two different levels of granularity that complement each other: (a) The *method declaration* level measures whether instances of a pattern are found within a single method declaration (the latter refers to the highest declaration in the type hierarchy that originally introduced the current method signature) or across method declarations (method-specific versus cross-method pattern). (b) The *code repository* level measures whether instances of a pattern are found in one or in multiple repositories (repository-specific versus cross-repository pattern). Knowledge about the generalizability of patterns is important for judging the versatility of the pattern in later applications.

4.4.4 Limitations

The concrete patterns we learn from our dataset may not be representative for all API usage instances responsible to complete a specific task. For example, a certain API usage might often be distributed over multiple methods, while our benchmark learns pattern occurrences contained within a single method declaration.

4.5 Pattern Types Benchmark (PTBench)

Following the idea of automated benchmarks, we facilitate the task of learning and comparing different pattern types on our dataset, with an automated experiment pipeline. We call this pipeline PtBench. The goal of PtBench is (1) to automate as much as possible of the experimental setup presented in Section 4.4, (2) to automatically compare different pattern type representations, by supporting the addition of different metrics definitions. (3) to make benchmarking experiments reproducible and extensible. The pipeline also enables benchmarking with different or extended datasets, in the future. We make PtBench publicly available for future studies.

4.5.1 Data Representation

For each project, the dataset records the project name, the project website, and the project repository. We use the repository to uniquely identify a project, because this allows us to uniquely identify project versions using their respective revision ID. For each project version, the dataset records the revision ID, the relative path to the source files and class files.

For each pattern, we store the enclosing method declaration in the source code that we identified it from. We use this information to uniquely identify a pattern.

4.5.2 Benchmark Automation

PtBench automates many of our evaluation steps including: (1) the transformation of source code into a stream of events, (2) the patterns' threshold values calculation, (3) running the episode mining algorithm to identify patterns, (4) filtering partial-order patterns, and (5) evaluating the learned patterns according to the three metrics defined in Section 4.4.3. These describe the main pipeline steps of PtBench we implemented to facilitate our evaluations and to enable easy replication of our experiments.

4.5.3 Reproducibility and Traceability

We publish PtBench[3] and encourage others to use the automated pipeline and contribute by experimenting with additional datasets and new evaluation metrics definitions, to conduct and repeat experiments and to extend the benchmark itself.

For legal reasons, we do not include source code or binaries of target projects in the benchmark itself, but instead provide links to the respective version-control systems and tooling that automates the retrieval of respective checkouts and their compilation. This minimizes the effort to collect the dataset and ensures access to the exact same versions of the projects. The results of checkout and compilation are stored locally, such that the respective data remains available for subsequent use. To make this as easy as possible, we build our benchmarking pipeline agnostic to the concrete dataset, such that the benchmarking experiments automatically consider new additions to the dataset.

Furthermore, we provide access to the patterns learned using PtBench, and address their target code.

4.6 Evaluation Results

This section presents the results of our experiments from the empirical comparison of the different pattern types. All experiments are performed with a frequency threshold of 345, and an entropy threshold of 0.72 (cf. Section 4.4.2). First, we present some statistics about the learned patterns, and then study them along the dimensions presented in Section 4.4.3.

4.6.1 Pattern Statistics

Here we analyze the learned patterns in terms of their size and number of API types they encode.

[3] http://www.st.informatik.tu-darmstadt.de/artifacts/patternTypes/

Table 4.1: Expressiveness results per configuration pair

	(*POC*, *SOC*)	(*NOC*, *POC*)	(*NOC*, *SOC*)
exact	858	248	0
subsumed	(260;346)	(716;986)	(853;1204)
new	116	17	128
Total	(1,234;1,204)	(981; 1,234)	(981;1,204)

Pattern Size refers to the number of events in a pattern. Our approach learns patterns with up to 7 events in each configuration. The number of patterns learned decreases for larger pattern sizes with the same ratio in each configuration. Almost all mined patterns (97%) involve 5 events or less. The result matches the intuition that it is less probable that many developers write large code snippets in exactly the same way.

API types within a pattern reflects the number of API types a pattern encodes interactions for. In all the patterns learned, 75% involve interactions between events from multiple API types (across configurations). Only 28% of the patterns with $2-4$ events involve interactions between events from a single API type. All patterns with 5 or more events involve multiple API types. The maximum number of API types involved within a pattern is 5 types, where patterns involving two API types make the majority (40%).

4.6.2 Expressiveness

Expressiveness quantifies the richness of the language corresponding to a pattern type, whose grammar rules are the mined patterns. Table 4.1 shows the expressiveness metric results. For each configuration pair `(c1,c2)`, *Total* shows the total number of patterns learned by `(c1,c2)` respectively.

POC vs. SOC These configurations learn 858 equal patterns, which implies that out of $1,234$ patterns learned by *POC*, 70% are sequences and only 30% of them include partial-order between events.

Observation1: Most of the API usage patterns define in the wild strict-order between events (70%), while the other 30% abstract over different API usage variants.

Furthermore, `subsumed(`*POC*`,` *SOC*`) is (260;346)`, i.e., 260 partial-order patterns learned by *POC* subsume 346 sequences learned by *SOC*. The 260 partial-order patterns encode 572 different sequences, i.e., the 346 sequences mined by *SOC* plus 226 others. Recall that multiple sequential-order patterns can be represented by a single partial-order pattern.

Finally, `new(`*POC*`,` *SOC*`)` is 116, meaning that for the events included in 116 partial-order patterns, there are no sequences learned by *SOC*. The 116 partial-order patterns encode 308 sequences of events that individually do not occur often enough in source code. For this reason, *SOC* does not mine them. On the other hand, *POC* represents different variants of sequences for the same set of events in a single pattern, which increases the partial-order pattern occurrence and makes it match the frequency threshold.

From these results, we can conclude that all patterns learned by *POC* represent a superset of the patterns learned by *SOC*.

Observation2: The API usage specifications encoded by partial-order patterns fully represent the specifications encoded by sequential-order patterns. Furthermore, they learn 116 additional patterns of events for which sequence mining cannot learn any sequence.

NOC vs. POC As shown in Table 4.1, `exact(`*NOC*`,` *POC*`) = 248`, which means that 20% of the patterns learned by *POC* are exactly the same as the ones learned by *NOC*. Recall that no-order patterns are mined in *POC* when the involved events occur often enough in either order.

Observation3: In 20% of the cases, partial-order patterns encode events that occur in either order in the wild.

Furthermore, `subsumed(`*NOC*`,` *POC*`)` is (716; 986), i.e., 716 no-order patterns learned by *NOC* subsume 986 patterns learned by *POC*. Note that one no-order pattern simplifies several partial-order patterns by removing order information.

Finally, `new(`*NOC*`,` *POC*`)` is 17, i.e., 17 patterns learned by *NOC* include events for which *POC* does not learn any pattern. These patterns are missed by *POC* because either: *(a)* none of the sequences between the events occur frequently enough, recall that sequences are a special case of partial-order patterns, and/or *(b)* there is not enough evidence in the source code that events occur frequently enough in either order (specified by entropy threshold).

From these results we can conclude that no-order patterns represent a superset of partial-order patterns.

NOC vs, SOC Table 4.1 shows that *NOC* and *SOC* learn 0 equal patterns, which is obviously the case, since *NOC* learns only set of events and *SOC* learns only sequences of events, i.e., there cannot be any overlap between the patterns learned by these two configurations. We find that `subsumed(`*NOC*`,` *SOC*`)` is (853; 1,204). In other words, all sequential-order patterns can be subsumed by 853 no-order patterns. Note that multiple sequential-order patterns can be simplified into a single no-order pattern by removing order constraints.

Finally, `new(`*NOC*`,` *SOC*`)` is 128, i.e., for 128 patterns learned by *NOC* there are no sequences mined by *SOC*. None of the sequences between these events occur frequently enough in the source code.

Observation4: No-order patterns represent all sequential-order patterns; furthermore, the no-order configuration learns 128 additional patterns for which sequential-order configuration could not learn any sequences.

Analyses of the Results To recap, sequence mining misses sequences of events which are captured by partial and no-order patterns. To understand what code structures they represent, we explored mined patterns and found examples that explain this phenomenon in the source code of Graphical User Interfaces (GUI). Using a GUI component typically requires to call its constructor first, but the order in which properties like color or size are configured is irrelevant. A miner thus finds many UI code examples with high variation and low support of each individual example. This reveals two disadvantages of the sequential-order miners. *First,* if the individual support for each variant of the GUI component usage is high enough, then redundant patterns will be identified, one sequence for each variant. *Second,* if the target threshold is not met by one or more sequence variants, the corresponding sequence pattern will be missed. In the same situation, each variant would count as support for patterns with more abstract representation such as partial and no-order, which thus may pass the threshold more easily. When compared with each other, partial-order patterns can preserve order information, which is missed by no-order patterns.

4.6.3 Consistency

Based on the results in Section 4.6.2, one may conclude that no-order patterns define a richer language compared to the other two types. The question raises: Why should one use expensive mining approaches (sequence or partial mining), if we can learn a richer language from source code using less computationally expensive mining approaches such as frequent item-set mining? However, this would be a valid conclusion, only if the words in the language mined by *NOC* are valid, i.e., the order between events in a pattern does not really matter. To analyze this, we investigate the consistency of the mined sequential and partial-order patterns with co-occurrences of events in code.

The consistency metric is a ratio that ranges in $]0.0, 1.0]$, as defined in Equation 4.1. The higher the consistency ratio, more important is the order information defined in p to the correct co-occurrence of events. If p has this value close to 1.0, it means that almost every time the events in p co-occur, they co-occur with the specific order defined in p, implying that the order information is very crucial for p. If p has this value close to 0.0, means that in very few cases the events in p co-occur in the order as defined in p, implying that the order information is not really important and p can be easily represented by a no-order pattern.

Our results reveal high consistency in sequential (avg. 0.9) and partial-order patterns (avg. 0.96). This suggests that order information encoded in both sequential and partial-order patterns is crucial for the correct co-occurrences of events in the wild, and simplifying them into no-order patterns will result in losing important order information between events.

Observation5: Partial and sequential-order mining learn important order information regarding co-occurrences of events in the wild.

Table 4.2: Code repository generalizability level for different configurations and pattern sizes

	Patterns		**2 events**		**3 events**		**4 events**		**5 events**		**6+ events**	
Config	Total	General	Total	General	Total	General	Total	General	Total	General	Total	General
POC	1,234	594 (48%)	573	472 (82%)	283	106 (38%)	212	15 (7%)	122	1 (1%)	44	0 (0%)
SOC	1,204	561 (47%)	562	458 (82%)	270	92 (34%)	206	10 (5%)	122	1 (1%)	44	0 (0%)
NOC	981	572 (58%)	531	445 (84%)	226	108 (48%)	132	17 (13%)	70	2 (3%)	25	0 (0%)

4.6.4 Generalizability

In this section, we present the generalizability metric results on two granularity levels as explained in Section 4.4.3: method declaration and repository level.

Method Declaration Our results empirically show that most of the patterns learned (98%) by each configuration, are used across method declarations. This means that the patterns learned generalizes to different implementation tasks, and are not tied to a specific task or context.

Observation6: Most of the patterns learned find applicability to a large variety of implementation tasks.

Next we analyze if the patterns learned are used by multiple developers, or if they represent specific coding styles for a given repository and its developers.

Code Repository Table 4.2 shows our results for different configurations and pattern sizes. The column *Patterns* shows the total number of patterns, and the absolute number and percentage of general patterns learned by each configuration. The next columns show the same information as *Patterns*, but for different pattern sizes, where the last column (6+ events) shows the information for patterns with 6 and 7 events.

Our results show that the patterns learned by *POC* and *SOC* have almost the same percentage of generalizability (48% vs. 47%), regardless of their size. This means that more than half the patterns mined by each configuration are learned from API usages from the same repository. While such repository-specific patterns are useful to the developers of that particular repository, they may reflect a very specific way of using certain API types, which may not be useful to a general set of developers.

As the table shows, *NOC* learns slightly more general patterns (58%). However, recall that these more general patterns come at the cost of missing order information between events.

Observation7: No-order patterns tend to be more generalizable (58%) compared to sequential and partial-order patterns (47% and 48%), which tend to be over-specified due to the order constraints they encode.

We analyzed the patterns learned exclusively by *POC* (recall Table 4.1) and found that 114 out of 116 patterns are general patterns used across repositories. To find out why most of the patterns learned exclusively by POC are general patterns, we check if there is any relation between generalizability and pattern-order. We find that strict-order patterns (`exact(`POC`, `SOC`)`) are less generalizable (37%) compared to patterns that contain partial-order between events (*subsumed* - 62%, and *new* - 98%). This confirms our hypothesis that there is a relation between generalizability and pattern-order. Furthermore, most of the patterns (90%) learned exclusively by POC include method calls only from the standard library, which further explains their re-usability across repositories.

Table 4.2 shows that across configurations, the percentage of general patterns learned is higher for smaller patterns, and significantly decreases for larger patterns. Furthermore, for patterns with 6 and 7-events, we learn only repository-specific patterns. Specifically, around 70% of general patterns (independent of the configuration) are 2 and 3-event patterns. Most of the patterns with 4-events or more are repository-specific patterns. This makes sense since the probability that multiple developers with different coding styles and different application domains writing a similar and long piece of code is very low.

Observation8: Small code patterns of 2 and 3 events are more generalizable compared to larger code patterns of 4 or more events that mainly encode constraints of API usages from a single repository.

We further analyzed the repository-specific patterns and found that 93% of them are learned from testing code, and they include API types that refer to an old version of a common assembly that is used in no other repository. Filtering out testing code may help mining algorithms learn only general patterns. An empirical validation of this hypothesis, however, needs to be performed in the future.

Remark: For the sake of completeness, we experimented with other threshold values (frequency and entropy), and analyzed the generalizability of the patterns across repositories. The results we received did not show higher generalizability ratios in neither of the configurations, compared to the ones presented above. This confirms the correctness of the threshold values selected as presented in Section 4.4.2.

4.7 Implications

Based on the evaluations results in Section 4.6, we derive the following implications:

Implication 1 (derived from Section 4.6.1): Mining techniques based on frequency occurrence of source code in code bases are unlikely to learn large code patterns (more than 7 method calls using our concrete parameters), since it is less probable that developers write large code snippets exactly in the same way. If the main goal is to learn large code patterns, then other techniques need to be considered.

Implication 2 (derived from Section 4.6.1): Code analyses techniques should consider interactions between objects of different API types, while extracting facts from source code. Even though these analyses are expensive since data-flow dependencies need to be considered, they are important in mining relevant patterns from source code.

Implication 3 (derived from Observation1 and Observation5): While covering a good amount of usages seen in source code, sequential-order mining may lead to false positives in applications such as misuse detection. For example, if the pattern is $a \rightarrow (b, c)$, but a strict-order pattern has only learned $a \rightarrow b \rightarrow c$ and the code written by the developer is $a \rightarrow c \rightarrow b$. On the other hand, while no-order mining might seem to learn a larger variety of API usages in source code, it might result in false negatives in such applications. Following the same example, the developer might have written $b \rightarrow a \rightarrow c$, and a no-order pattern cannot detect that b and c should occur strictly after a. We can conclude that, partial-order mining learns better API usage patterns for such applications.

Implication 4 (derived from Observation2): Partial-order mining might be more appropriate for learning API usage patterns in applications such as code recommendation since multiple sequences can be represented by a single partial-order pattern, decreasing the total number of patterns that need to be part of the model. In sequence mining, multiple patterns need to be recommended to the developer for the same set of events and might even risk missing valid sequences if they do not occur frequently enough in the training source code.

Implication 5 (derived from Observation5): Before deciding which mining approach to use in a specific application, developers need to know their trade-offs in terms of order information and computation complexity. Sequential and partial-order mining are computationally expensive approaches but learn important order information about the co-occurrence of events in a pattern, while no-order mining approaches do not require expensive computations but on the other hand do not learn any order information about the co-occurrence of events in a pattern.

Implication 6 (derived from Observation8): If the main goal is to learn large code patterns ($4 - 7$ events), then recommenders should focus on a repository-specific mining approach and produce catered recommendations to the repository's developers. However, if the goal is to learn general patterns that can be used by many developers, then researchers should know that they might end up mining small patterns (2 to 3 events).

4.8 Threats to Validity

4.8.1 Internal Validity

We generate the event stream based on static analyses, not on dynamic execution traces. Even though this may not represent valid execution traces, it does represent how the code is written by developers. In this part of the thesis, we focus on learning code patterns to represent source code as it is written in code editors. Also, our event stream considers only intra-procedural analysis since we are interested to learn patterns that occur within methods. Using inter-procedural analysis might affect our results.

The episode mining algorithm learns only *injective* episodes, where all events are distinct, i.e., the algorithm does not handle multiple occurrences of the same event in a pattern. For example, method invocations: `IEnumerator.MoveNext()` or `StringBuilder.Append()` are usually called multiple times in the code. The patterns we learn contain a single instance of such events. While this is a limitation, it is also an advantage in terms of pattern generalizability. Specifically, the mined pattern would not have a strict number of occurrences that would lead to mismatches between it and another valid code snippet that has a different number of occurrences.

The algorithm relies on user-defined parameters: *frequency* and *entropy-thresholds*. While the configuration parameter depends on the type of patterns one is interested in, deciding on adequate frequency and entropy thresholds is not an easy task, and which significantly affect the results. We mitigate this threat by empirically evaluating the thresholds and choosing the best combination of frequency and entropy thresholds for the given data set (cf. Section 4.4.2).

The episode mining algorithm is available only in a sequential (non-parallelized) implementation, hence is inefficient. However, this thesis does not work towards improving the performance of episode mining algorithm per se, but rather uses it as a baseline for comparing the different configurations by automatically learning different code structures from code repositories. This limitation can be improved by parallelizing the algorithm's implementation.

4.8.2 External Validity

In this part of the thesis, we do not learn patterns for project-specific API types. Extracting code patterns for project-specific API types can still be achieved using the episode-mining algorithm as we do. Comparing project-specific patterns between different types of projects is an interesting task for future work.

We learn code patterns only for *method declarations* and *invocations*, excluding all other code structures such as *loops*, *conditions*, *exceptions* etc. This is because the focus on this part of the thesis is on automatically learning different code representations (sequential, partial, and no-order), instead of specifically learning complex patterns that include all code structures. Since learning code patterns while considering other code structures is important for supporting certain development tasks, we plan to enrich the code patterns that we learn with additional code structures in future work. This requires modifying our event stream generation, which is an engineering task rather than a conceptual limitation.

Finally, we analyze the trade-offs between different pattern types using the same set of code repositories written in the same programming language. We also use a single learning algorithm that we configure to produce different pattern types. We use an established data set of 360 repositories that have over $68M$ lines of source code to ensure that we analyze large amounts of code and different coding styles. However, we cannot generalize our results beyond our current dataset and learning algorithm.

The evaluations provided can be a good starting point for further investigations and a practical baseline for developers dealing with API usages. Additionally, the available benchmark could be a useful baseline for researchers exploring properties of API patterns. However, even if promising, this is the first empirical attempt to compare API pattern types in terms of effectiveness in representing API usages in the wild, and further investigations would be necessary before declaring the outcomes as generic conclusions.

4.9 Discussion

In this part of the thesis, we present the first benchmark for analyzing the trade-offs between three pattern types (sequential, partial and no-order) with respect to real code. Our approach consists of three steps: the transformation of source-code into a stream of events, the adaptation of an event mining algorithm to the special context of pattern mining for software engineering, and filtering of the resulting patterns.

Our empirical investigation shows that there are different types of patterns learned in code repositories. While there are tradeoffs between pattern types in terms of *expressiveness*, *consistency* and *generalizability*, they are comparable in terms of the patterns size and number of API types. Our results empirically show that the *sweet spot* are *partial-order* patterns, which are a superset of *sequential-order* patterns, without losing valuable information like *no-order* patterns. Partial-order mining finds additional patterns that are not identified by sequence mining, and which are used by a larger number of developers across different code repositories. Compared to no-order mining, partial-order learns a smaller percentage of cross-repository patterns (58% vs. 48%), due to the order constraints between events within a pattern. Evaluation results show that all three configurations end-up learning only *repository-specific* patterns for pattern sizes with 6-*events* or more. Furthermore, our results empirically show the *consistency* of order information in sequential and partial-order patterns: on average 90% and 96% respectively. This means that even though no-order patterns represent the most generalizable type, they cannot substitute sequential and partial-order patterns due to the loss of order information.

Our findings are useful indications for researchers who work with code patterns in applications such as code recommendation and misuse detection.

5 On the Impact of Order Information in API Method Call Misuses

In Chapter 4, we presented the first empirical comparison of API pattern types in representing source code in the wild. In this study, we identified several implications, that help researchers to build better applications based on API usages, such as:

- Code analyses techniques should consider interactions between objects of different API types, while extracting facts from source code. Even though these analyses are expensive since data-flow dependencies need to be considered, they are important in mining relevant patterns from source code.
- While covering a good amount of usages seen in source code, sequential-order mining may lead to false positives in applications such as misuse detection. Sequence mining might miss learning valid sequences of method calls because of their low occurrences. As a result if two method calls can occur in either order and one of these sequences is missed during the mining process, then its usages in source code might be identified as misuses (false positives). At the same time, no-order patterns might result in false negatives in such applications due to missing order information, for example when two method calls occur in the wrong order in source code.
- Before deciding which mining approach to use in a specific application, developers need to know their trade-offs in terms of order information and computation complexity. Sequential and partial-order mining are computationally expensive approaches but learn important order information about the co-occurrence of events in a pattern, while no-order mining approaches do not require expensive computations but on the other hand do not learn any order information about the co-occurrence of events in a pattern.
- If the main goal is to learn large code patterns (4 - 7 events), then recommenders should focus on a repository-specific mining approach and produce catered recommendations to the repository's developers. However, if the goal is to learn general patterns that can be used by many developers, then researchers should know that they might end up mining small patterns (2 and 3 events).

Our empirical evaluation results (presented in Section 4.6.4) showed that even though we applied the episode mining algorithm in a cross-repository setting, most of the learned patterns were repository-specific. Therefore, in this part of the thesis, we go one step further and use the episode mining algorithm to mine patterns directly in a per-project setting, and analyze the effectiveness of the different pattern types in the concrete application domain of misuse detection. One hypothesis is that individual projects contain too few usage examples to mine good patterns [13]. This needs to be properly addressed in this part of the thesis by optimizing the threshold values on a per-project setting, as presented in Section 5.3.2. Incorrect usages of an API, or *API misuses*, are violations of usage constraints of the API. API misuses lead often to software crashes, bugs, and vulnerabilities in the source code.

To provide a fair setting, we extend the benchmark presented in Chapter 4 with the following components:

- A second data set, Java data set (MuBench [11]) by providing support for yet another programming language. The Java data set is used as the ground-truth to compare the different pattern types within the application context of misuse detection. To the best of our knowledge, there does not exist an equivalent data set for C#, which motivated us to provide support for another programming language in our benchmark (PtBench).

- A misuse detector (EmDetect) in order to evaluate the effectiveness of the different pattern types and the validity of the above implications derived from the comparison of pattern types, within a real application context. We compare the pattern types in terms of both, precision and recall.

In Section 5.1, we discuss these problems in more detail and sketch how we address them with the new API-misuse detector that we introduce in this part of the thesis. In Section 5.2, we present our misuse detector EmDetect. EmDetect encodes API usages as episodes. EmDetect employs episode mining algorithm to mine patterns and a specialized graph-matching strategy to identify (violating) occurrences of patterns. Both components consider code semantics of API usages to improve the overall detection capabilities. On top, EmDetect uses an empirically optimized ranking strategy to effectively report true positives among its top-ranked findings. We assess and compare the performance of EmDetect and four other state-of-the-art detectors on this extended benchmark. In Section 5.3, we present the evaluation setup by describing: the Java data set, analysis of the threshold values used by the mining algorithm, and the experiments performed to evaluate our misuse detector. In Section 5.4, we present the results of our evaluations, by comparing the performance of the different patterns types within the special context of a misuse detection. In Section 5.5, we discuss the extensibility and reusability of PtBench. The benchmark eases the extension with other datasets (programming languages), the integration of new metric definitions and of further application contexts, to increase generalizability of experiment results and encourage cross-pattern types comparison. In Section 5.6, we discuss the threats to the validity of our results and in Section 5.7, to conclude this part, we provide an overview over related work on API-misuse detection.

5.1 Background and Motivation

Monperrus *et al.* [92] report that issues related to missing method calls are prevalent in bug trackers, forums, newsgroups, commit messages, and source-code comments. Wasylkowski *et al.* [158] investigate problems where methods are called in the wrong order. These facts show the high interest in developing approaches that can automatically detect such method call misuses during development time, and offer the right fixes (recommendations) to developers.

The main goal of this chapter, is to present a new misuse detector based on the episode mining algorithm (EmDetect), which is able to automatically discover different code representations from source code, in contrast to other state of the art approaches that are based on program analyses that pre-define the code representation and simple aggregation algorithms for learning the patterns (Table 2.1). We evaluate the quality of the learned patterns within the application context of misuse detection, and compare their performance with respect to other state of the art approaches. EmDetect employs the same matching and ranking strategy as the ones used by MuDetect [13]. This allows for a direct comparison of the performance of two detectors that differ only on the static analyses and learning algorithm used: one based on detailed static analyses and pre-defined code representation (MuDetect), and the other based on a simple stream of events extraction and automatic learning of different code representations (EmDetect). For the sake of completeness, we also compare EmDetect with the other detectors used in MuDetect evaluations: GrouMiner, Jadet, Tikanga, and DMMC in terms of both precision and recall. We introduce these detectors in the following.

MuDetect [13] encodes API usages as API-Usage Graphs (AUGs) that captures all usage properties based on data and control flow analyses. Node represent data entities, such as variables and actions, for example method calls, conditions, iterators and exceptions; edges represent control and data flow between the entities and actions represented by nodes. MuDetect employs a code-semantic-aware, greedy frequent-subgraph-mining algorithm to mine the patterns.

GrouMiner [110] represents usages as directed acyclic graphs that encode method calls, field accesses, and control structures as nodes and control-flow and data-flow dependencies among them as unlabeled edges. GrouMiner uses sub-graph mining to find patterns and then detects violations of these patterns as missing nodes. It detects missing method calls, as well as missing conditions on the granularity of a missing branching or loop node.

Jadet [158] encodes the transitive closure of the call-order relation in each usage as pairs of the form `m()` $\prec$ `n()`. It uses Formal Concept Analysis [41] to identify violations, i.e., rarely missing call order pairs. Tikanga [157] builds on the same algorithm as Jadet , but encodes usages using temporal properties (CTL). Both detectors detect missing calls. However, Jadet cannot detect violations of patterns with only two calls, because it works on multiple call pairs, since this would be a usage with a single call, which cannot be encoded using call pairs. Tikanga can detect such violations.

DMMC [92] encodes usages as sets of methods called on the same receiver type. It identifies violations by computing, for every usage, the ratio between the number of equal usages and the number of usages with exactly one additional call. Intuitively, a violation should have only few exactly-similar usages, but many almost-similar usages. DMMC detects misuses with exactly one missing method call.

These four detectors reveal several problems that result in low recall and precision. In the following, we give a brief description of these problems and how we mitigate them.

5.2 A New Detector

In this section we explain how we use the learned API usage patterns for detecting potential API misuses in source code. For this purpose, we build a new API-misuse detector, EmDetect as follows:

(1) We use the episode mining algorithms introduced in Section 4.3, for learning the different pattern types. Episode mining takes as input a stream of events (extracted from source code), and outputs episodes (patterns) as partial order set of the extracted events.

(2) Given the graph representation of the patterns we learned (Figure 4.3), we adapt the algorithm used by MuDetect for detecting and ranking potential API misuses found in source code. This allows us to fairly compare the subgraph mining approach used by MuDetect which transforms source code into a detailed graph representation using data and control flow analyses, with the episode mining approach which transforms source code into a simple stream of events.

We subsequently introduce EmDetect's components, one at a time.

5.2.1 Pattern Mining

The mining algorithm and pattern generation steps are the same as described in Section 4.3.2. In order to optimize the patterns outputted by the episode mining algorithm to the special context of misuse detection, in addition to the pattern filtering step introduced in Section 4.3.2, we apply also the following filter: We ignore sub-patterns, e.g., patterns that are part of some other larger patterns. Given that pattern mining is an a-priori-based algorithm, a sub-pattern $(a \rightarrow b)$ might be part of another pattern $(a \rightarrow b \rightarrow c)$, if the later occurs frequently enough according to the frequency and entropy thresholds. The $(a \rightarrow b)$ constraint is a redundant constraint already included into the $(a \rightarrow b \rightarrow c)$ constraint, which is not required for the purpose of misuse detection and that's why we filter it out.

5.2.2 Detecting and Ranking API Misuses

Given the graph representation of the patterns we learn, we use the same algorithm as MuDetect for detecting and ranking potential API misuses found in source code. The detection algorithm takes as input the set of learned patterns, the target source code that we want to analyze for potential API misuses, and outputs a ranked list of potential misuses. In a nutshell, the algorithm works as follows.

(1) The detection algorithm checks and discovers for each pair of a pattern and target source code, full occurrences (instances) and partial occurrences (potential misuses).

(2) Potential misuses that are subgraphs of instances of another pattern are filtered out, since they represent alternative correct usages of the same API. Hence they don't represent an API misuse.

(3) After identifying all potential misuses in the target source code, the detection algorithm ranks the findings using different ranking strategies. Some of these ranking strategies come from the literature [66, 92, 110, 149, 150, 157, 158], and others are generated as combinations of the individual ranking factors by multiplication. The following ranking factors are considered: pattern support, number of pattern violations, the pattern uniqueness factor, violation support and the violation overlap. Since it is unclear which of these strategies is useful, they are evaluated empirically.

(4) To avoid reporting duplicate misuses (i.e. usages that violate alternative correct usages of the same API), the algorithm filters out misuses involving a method call that is part of another misuse listed with a higher rank.

More details about the detection algorithm can be found in the work by Amann et. al. [13].

As mentioned before, the main contribution in this part of the thesis is not to build a better misuse detector, but to show instead that by using (1) a simple AST parser and (2) an advance machine learning algorithm that is able to automatically discover latent knowledge (different pattern representations) from source code, we are able to obtain comparable results compared to state of the art based mainly on (1) detailed program analyses for extracting domain knowledge code artifacts and (2) simple aggregation algorithms for abstracting these information. For this reason, to make the comparability with the state of the art as close as possible, as explained above, EmDetect adopts the same detection and ranking algorithms as MuDetect. MuDetect is based on AUGs, which are acyclic connected graphs, while EmDetect is based on episodes that in the partial and no-order configurations output disconnected graphs. For this reason, before applying MuDetect detection algorithm, we need to convert the learned patterns by EmDetect into connected graphs representation (AUGs). How the conversion is handled in the different configurations of EmDetect is explained in the following:

- Sequential-order configuration is straightforward, since sequences are by definition connected.

- Partial-order configuration: (a) for disconnected 2-event patterns, we generate both possible sequences and remove the disconnected version. (b) For disconnected larger patterns, containing more than 2-events, slightly adapt the filtering step explained in Section 4.3.2, by maintaining the most abstract connected patterns, so that from the set of patterns representing the same events, all the kept patterns are connected and represent all the other (filtered) patterns.

- No-order configuration, we skip this configuration for the evaluations in this chapter of the thesis, because of what we found in **Observation5**.

5.3 Evaluation Setup

This section describes the data set we use, presents the analyses of the frequency and entropy thresholds for the episode mining algorithm, and presents the setup we use to assess EmDetect ability to detect API misuses in the different mining configurations. Our main goal is to evaluate EmDetect's precision and recall, especially compared to existing detectors, in order to understand whether either a simple representation of source code, such as a stream of events, is effective in practice when using more sophisticated machine learning approaches.

5.3.1 Dataset

For evaluating EmDetect in terms of both precision and recall, we need an annotated data set of correct and incorrect API usages. To the best of our knowledge, such a data set does not exist for C# code, that's why we need to extend our benchmark (PtBench [30]) to support yet another programming language for which there exists a ground-truth of known API misuses. We chose to use MuBench [11] as our ground-truth, which contains API method call misuses with examples of correct usages, derived from the fix of the corresponding misuse. The API method call misuses come from real-world Java projects. Furthermore, MUBench comes with MuBenchPipe [12], a public automated benchmarking pipeline built on top of MUBench. MUBenchPipe reveals us from the burden of preparing the target projects and executing the detector, since everything is already integrated into the automated, publicly available pipeline.

In this chapter, we compare EmDetect against the five detectors: MuDetect, Jadet, GROUMiner, Tikanga, and DMMC. As the ground-truth for the experiments, we use MuBench, a dataset of 191 API misuses. For simplicity, we refer to this dataset as MuBench throughout this chapter of the thesis.

5.3.2 Threshold Analyses

The episode mining algorithm uses two thresholds: *frequency* and *entropy*. The threshold values directly impact the number of patterns learned (higher threshold values means stronger evidence in the source code that a given pattern occurs), and as a consequence also the performance of the misuse detector (according to the patterns learned, the detector may or may not identify misuses in the source code).

In the application context of misuse detection, we empirically evaluate the effect of frequency and entropy thresholds on the performance of EmDetect in terms of: *(1)* The number of misuses detected, based on the ground truth of known method call misuses that we have on the Java projects. *(2)* The performance of the ranking algorithm (presented in Section 5.2.2) on ranking true positives on top of the list of findings.

Initial Analyses Given that we have to analyze the findings of the detector manually, we perform our analyzes for the frequency and entropy thresholds on one Java project from MuBench. For this we chose the one with the highest number of the method call misuses. After fixing the Java project (*initial*) on which we perform our threshold analyzes on, we choose an arbitrary value for the frequency threshold, and analyze the effect of different entropy thresholds on the performance of the detector. After defining the optimal entropy threshold, we repeat our analyzes to study the effect of the different frequency values. Our analyzes reveal an optimal frequency threshold of 20, and entropy threshold of 0.4 for the *initial* project we perform our analyzes on.

Automating for different projects Since different projects have different sizes (number of events), the frequency threshold highly influences the number of patterns learned in each project. For this reason, we decided to automate the calculation of the frequency threshold according to the project sizes. For this we considered the total number of events, the number of unique events, and the average occurrences of events in each of the projects. According to our analyzes, the best function for this calculation resulted the one that compares a *target* project with the *initial* project, on the average occurrence of events. The function used for calculating frequency threshold in every project is the following:

$$frequency = \left(1 + \frac{avg.Target}{avg.Initial}\right) * InitialFreq \tag{5.1}$$

, where *avg.Target* and *avg.Initial* is the average occurrence of events in the target and initial project respectively, and *InitialFreq* is the frequency threshold used in the initial project. The output of this function we round up to the 5th closest integer, for example if the function outputs either 22.2 or 24.8 they are both rounded up to 25.

5.3.3 Experimental Setup

We evaluate EmDetect performance on both precision and recall. For the evaluations, we use an entropy threshold of $e = 0.4$, and varying frequency threshold according to project sizes as presented in Section 5.3.2. We run the experiments using MuBenchPipe [12], a public automated benchmarking pipeline built on top of MuBench [11], containing the ground-truth data set. MuBenchPipe facilitates preparing the target projects from MuBench. Executing the detectors on them, and collecting result statistics about the detectors' performance we manually reviewed the detectors' findings. Since the patterns mined by the episode mining configurations consist only of events that correspond to method declarations and invocations, we run EmDetect on a subset of projects from MuBench that contain API method call misuses, letting it mine patterns and detect violations on a per-project basis. Since we compare the performance of EmDetect with state of the art detectors presented in Section 5.7, we apply a second filter on the projects to select the ones that are also used by Amann et. al. in [12]. This left us with a total of four Java projects.

To evaluate the precision and recall of EmDetect, we conduct two experiments, namely P to measure precision, and R to measure recall. For the other detectors, we use the best configurations as reported in the respective publications. We introduce the experiments in more detail in the following paragraphs.

Precision We run the detector on the four projects from MuBench, to mine patterns and detect violations on a per-project basis, and on the different mining configurations. Since EmDetect reports several hundreds of violations, reviewing all violations of all mining configurations and on four projects is practically infeasible. Therefore, we reviewed the $top - 20$ findings per configuration on each of the selected projects, as determined by the ranking algorithm to identify true and false positives. The new true positives found that are not part of the ground-truth, are candidates to be included in MuBench.

Recall We run EmDetect on four projects from MuBench (containing API method call misuses and used in [12]), and on the different mining configurations. Following the same evaluation logic as in [12], we detect violations on a per-project basis. Then, we manually reviewed all potential hits, i.e., all findings in the same method as a known misuse. As the ground truth, we use the known API method call misuses from MuBench. We report the number of misuses identified by each of our mining configurations. This gives us the recall of the detector with respect to known misuses and, at the same time, crosscheck which of the mining configurations' findings are also identified by the other configurations and/or by the state of the art detectors.

5.4 Evaluation Results

In this section, we present the results of our experiments in comparing sequential and partial-order patterns within the application context of API method call misuse detection. We use our misuse detector (EmDetect) to compare the pattern types in terms of both precision and recall. The experiments in this section are performed on the Java data set, using MuBench [11] as a ground-truth of correct and in-correct API usages for evaluating the detectors' performance. For the sake of completeness, we compare the performance of EmDetect also with the other 4 misuse detectors studied by Amann et. al. [12]. For evaluating the detectors, we consider the same set of projects as used in [12] and select the ones that contain API method call misuses. This let us with a total of 4 Java projects to perform our evaluations on: `bcel`, `chensum`, `jigsaw` and `testing`. All experiments ran on a *MacBook Pro* with an *Intel Xeon @ 3.00GHz* and *32GB of RAM.*

Table 5.1: Precision of the Detectors in Their Top-20 Findings.

	Precision		Recall		F_1
Detector	True Positives	%	Hits	%	%
$EMDetect_{POC}$	3	3.8	8	42.1	7
$EMDetect_{SOC}$	12	15	5	26.3	19.1
MuDetect	12	15	14	51.8	23.3
DMMC	2	3.3	3	15.8	5.5
Jadet	4	7.7	8	42.1	13
GROUMiner	3	3.3	7	36.8	6.1
Tikanga	2	5	2	10.5	6.8

5.4.1 Precision

The first part of Table 5.1 summarizes the results of measuring the detectors' precision in their $top - 20$ findings.

Observation 7.1: $EMDetect_{POC}$ reports 80 violations in the $top - 20$ findings in four projects. Among these violations, we find three true positives, two of which were previously unknown. This results in precision of 3.8%, which exceeds the precision of 2 of the detectors from the literature.

Observation 7.2: $EMDetect_{SOC}$ report 80 violations in the $top - 20$ findings in four projects. Among these violations, we find 12 true positives, 9 of which were previously unknown. This results in precision of 15%, which is the same as the precision for MuDetect and exceeds the precision of all the other detectors from the literature.

The two observations above show that $EMDetect_{SOC}$ performs better in terms of precision compared to $EMDetect_{POC}$, by ranking more true positives in the $top - 20$ findings. This comes due to: *(1)* higher number of patterns learned by *POC* compared to *SOC* as we found in **Observation2**, and *(2)* missing of the order information between some of the events in partial-order patterns. The higher number of patterns means that more false positives are ranked in the $top - 20$ findings, while the missing of the order information impacts the matching algorithm, which is based on nodes (method calls) and edges (order information).

5.4.2 Recall

For measuring the detectors' recall, we use 19 publicly available method call misuses from the four filtered projects from MuBench. The right part of Table 5.1 summarizes the results.

Observation 7.3: $EMDetect_{POC}$ identifies 8 out of the 19 known misuses, which results in recall of 42.1%. This result exceeds the recall of three out of five detectors from the literature, except for JADET with which it performs the same and MUDETECT which shows a better recall (51.8%).

Observation 7.4: $EMDetect_{SOC}$ identifies 5 out of the 19 known misuses, which results in recall of 26.3%. This result exceeds the recall of two out of five detectors from the literature.

The two observations above show that $EMDetect_{POC}$ performs better in terms of recall compared to $EMDetect_{SOC}$, by finding more known misuses from our ground-truth data set. This comes due to the fact that POC abstracts over several usages in the source code, which increases the patterns support. On the other hand SOC learns only sequences of method calls and, when a given sequence does not occur often enough, it is missed by the learning algorithm.

$EMDetect_{POC}$ correctly identifies three misuses that $EMDetect_{SOC}$ does not identify, and one misuse that none of the detectors from the literature nor $EMDetect_{SOC}$ identifies. $EMDetect_{POC}$ misses 8 misuses that one of the detectors from the literature finds. Three of these misuses are missed, because the projects contain few usage examples compared to the frequency threshold used by EMDETECT for the pattern mining algorithm. Four of these misuses are missed because they contain a missing call in case an exception occurs. Since EMDETECT does not handle exception conditions (it only identifies if a method is missing or not in the target code), it fails in identifying such cases. One of these misuses is missed due to the matching algorithm. Overall, the detectors identified 34 unique previously unknown misuses in experiment P.

While EMDETECT has higher recall than the other detectors, its recall is still low in absolute terms. We find that EMDETECT has on average 227.6 usages examples (median = 105) for APIs whose misuses it identifies, but only 38.6 examples (median = 11) for those it misses. There is a moderate correlation (Pearson's $r = 0.52$) between the number of examples and detecting a misuse. This supports the hypothesis that the target projects contain too few usage examples for some APIs.

5.4.3 Discussion

Our evaluation results in the application context of misuse detection show that EMDETECT_{SOC} performs better in term of precision by ranking true positives higher in the $top - 20$ findings, while EMDETECT_{POC} outperforms EMDETECT_{SOC} in terms of recall, since it is able to abstract over several API usages with low occurrence, making SOC fail in learning such patterns.

Compared to the other detectors from the literature, we can conclude that EMDETECT_{SOC} outperforms all of them in terms of precision by at least 2 times, and EMDETECT_{POC} performs better (DMMC, GROUMINER and TIKANGA) or the same (JADET) and worse when compared to MUDETECT in terms of recall. Depending on whether we give higher priority to either precision or recall, we can decide on the mining configuration to use, either POC or SOC. Our results also show that it is possible to outperform other detectors in the literature with a general purpose machine learning approach (episode mining) that does not require much domain-specific tuning.

EmDetect is able to identify wrong call order and missing method calls, but fails in identifying superfluous method calls. The false negatives caused by *superfluous* method calls cannot be detected by misuse detectors that search for missing elements. From the literature, DroidAssist [112] uses a *probabilistic approach* that might find superfluous method call, but the approach has never been evaluated.

5.5 Extension and Further Use

We design PtBench as an extensible automated benchmark, to facilitate not only our own study of API-pattern detectors presented in Chapter 4, but also future work. As the most important extension points we consider (1) extensions to the benchmarking dataset, to increase our confidence in the generalizability of the benchmarking results also to other programming languages, (2) integrating additional metrics for evaluating pattern types, and (3) integrating additional applications, to move further towards a comprehensive comparison of the pattern types.

5.5.1 Dataset Extensions

The dataset forms the basis for the experiments. Though ideally a benchmark dataset is a minimal representative sample, to the best of our knowledge, it is unclear how such a sample may be determined. Therefore, the best way to approximate representativeness is to extend the dataset by other projects, and/or source code of projects developed in other programming languages, to make the comparison of the pattern types as representative as possible within and across programming languages. For further technical details on how to extend PtBench with projects from other programming languages, we refer to the project website.[1]

5.5.2 New Metrics for Pattern Comperison

In Chapter 4 we presented three metrics on which we base on the comparison between the different pattern types we learn, namely: expressiveness, consistency and generalizability. PtBench is extensible in the meaning that other researchers are encouraged to add additional metrics definition on it for evaluating the learned patterns, and investigate differences between the pattern types considering the new dimensions.

5.5.3 Comparison of Pattern Types based on Applications

To achieve a broad empirical comparison of the different pattern types, it is crucial that we empirically evaluate their performance in other software engineering applications (i.e. code recommendation), in addition to misuse detection which is already covered here.

[1] http://www.st.informatik.tu-darmstadt.de/artifacts/patternTypes/

For technical details on how to integrate new applications into PtBench, we refer to the project website.

5.6 Threats to Validity

Internal Validity The episode mining algorithm learns only injective episodes, where all events are distinct, i.e., the algorithm does not handle multiple occurrences of the same event in a pattern. For example, method invocations: `IEnumerator.MoveNext()` or `StringBuilder.Append()` are usually called multiple times in the code. The patterns we learn contain a single instance of such events. While this is a limitation, it is also an advantage in terms of pattern generalizability. Specifically, the mined pattern would not have a strict number of occurrences that would lead to mismatches because of the difference in the number of occurrences. However, this is not the case for the other detectors from the literature that we compare EmDetect with. MuDetect stores information even for the exact code line where a known misuse occur. These differences have a positive impact on respectively the precision and recall of the other detectors compared to EmDetect.

The algorithm relies on user-defined parameters: frequency and entropy thresholds. While the configuration parameter depends on the type of patterns one is interested in, deciding on adequate frequency and entropy thresholds is not an easy task, which affect the results. We mitigate this threat by empirically evaluating the thresholds and automize the frequency threshold per project size (cf. Section 5.3.2).

We adapt the same violation detection and ranking algorithm developed for MuDetect. While the implementation details for these algorithms might be optimized for MuDetect, it might not define the best performance for the patterns mined by EmDetect. However, the goal here is not to build a perfect detector, but rather comparing the performance of a mining approach based on program analyses for encoding code semantics into a graph representation (MuDetect), with a more general mining approach that simply converts source code into a stream of events and automatically learns different code representations without a predefined format.

We reviewed the EMDetect' findings ourselves. For the other detectors, we used the results as presented in [13].

External Validity The study is subject to the limitations of MuBench dataset. We cannot generalize our results for other datasets, or the performance evaluations with regard to other misuse detectors, except of the ones included here.

5.7 Related Work

Helping developers identify API misuses has received much attention. As a matter of facts, most of the approaches (57%) presented in Table 2.1 are developed for the purpose of misuse detection. Some of these approaches learn API patterns across different projects and use the learned patterns for detecting misuses in the project of interest, and others learn project-specific API patterns and use them to detect potential misuses within the same projects.

Per-project API-Misuse Detectors In this chapter, we presented a comparison of our detector (EmDetect), to the other detectors that target Java: MuDetect, GrouMiner, DMMC, Jadet and Tikanga. MuDetect and GrouMiner are both based on a graph-based representation of API usages (GROUMs and AUGs), and use subgraph mining for learning the patterns. AUGs, in contrast to GROUMs, are directed acyclic multi-graphs that capture method calls, field accesses, `null` checks, and data entities as nodes and control/data dependencies among them as labeled edges. Control structures are encoded by control edges between the nodes representing the conditions and the nodes representing the controlled actions. The difference between these two representations is that while GrouMiner can detect a missing `if`, MuDetect can also tell what should be checked in the `if` condition. Additionally, AUGs encode exceptional, synchronized, and iterative control flow, and distinguish receivers from parameters, to differentiate between correct usages and misuses.

Jadet uses as well a directed graph-based representation of source code, where nodes represent method calls on a given object and edges represent control flows. From the graph representation, it derives a pair of calls for each call-order relationship. Object usages are then represented as a set of such pairs, which form the input to the miner for learning the patterns. The encoding of call-order relation, allows Jadet to detect missing method calls. It may also detect missing loops as a missing call-order relation from a method call in the loop header. However, it cannot detect violations of patterns that consist of only two calls, since such a patterns would be represented by only a single pair of method calls. Tikanga builds on the same algorithm as Jadet. It replaces Jadet call-order properties by general Computation Tree Logic formulae on object usages. It applies Formal Concept Analyses [41] to obtain patterns and detect violations at the same time. In difference to Jadet, Tikanga is able to also detect violations of patterns with only two method calls. DMMC is also specialized in detecting missing method calls, but instead of mining patterns, it rather computes a likelihood for every usage to be a potential misuse. DMMC is able to only detect misuses with exactly one missing method call.

Misuse detectors using frequency-based approaches for learning the API patterns, are unable to detect redundant methods calls as misuses, redundant invocations of method calls in the source code that are not required. Redundant method calls are also usually known as *code smells*. For the approaches presented in Table 2.1, only three of them are able to detect redundant method calls: PJAG12 [121], DroidAssist [111] and Salento [96]. In all three cases, this ability comes from patterns modeling object states, using method calls to signal state transitions. However, it is unclear whether and how the above three mentioned approaches can be extended to also cover misuses of other code elements, such as conditions, exception handlings and transitions.

Except of Java, many other misuse detector are developed for detecting misuses in other programming languages as well, such as [3, 66, 67, 126, 127].

Cross-project API misuse detectors Gruska *et al.* [44] evaluated JADET in a multi-project setting, where it simultaneously mined patterns and detected violations in a combined set of all usages from 6,000 projects. JADET mines any pattern with high support, even within a single project. From the reviewed violations, Gruska *et al.* published only 8% of them as true positives. Instead of mining patterns across-projects, CARMINER [150] and ALATTIN [149] mine target-specific patterns from examples retrieved via a code-search engine. This presents an alternative to the cross-project mining. In contrast to other approaches, where they usually mine most commonly used patterns first and then find deviations from them, Ammons *et al.* [14] learns the patterns iteratively from each client program. For learning the patterns, they analyze execution traces of API method calls sequences on pre-defined APIs. Given a client program, they randomly select an execution trace and mine a specification from it. An expert examines the specification and judges if it is correct or not. In case of correct specifications, they add it to the set of patterns already learn, otherwise they mark it as a buggy trace and randomly select a new execution trace till they find a correct one. The same procedure is followed for each client program. The approach proposed by Ammons *et al.* uses the set of already learned patterns to identify potential misuses to new client programs.

6 Conclusion and Outlook

In this thesis we investigate the advantages of machine learning in two dimensions. *First*, by using the same program analyses as defined in a state of the art approach (PBN [123]), and investigating if an approach that uses machine learning can find additional latent knowledge that was not possible to uncover before. For this purpose, we use a *Boolean Matrix Factorization* (BMF) approach. We show that BMF overcomes many of the drawbacks that come with simple clustering approaches, such as automatically discovering the number of clusters to represent the object usage space from source code, and identify corner cases (noise) in the data. To evaluate BMF performance, we use the PBN recommender, which is designed as an extensible inference engine for method completion. We replace the originally used canopy clustering with BMF, and compare the performance of both approaches in terms of model size, prediction quality and inference speed.

Second, we use an event stream mining algorithm that automatically learns different code representations (sequences, sets and partial-orders), without complex domain knowledge needed to encode a-priori. Designing a code representation that enables effective learning is a critical task that is often done manually for each programming language. The main idea behind episode mining is to represent source code as a stream of events by traversing its abstract syntax tree (AST). This allows a learning model to leverage the structured nature of code, while still considering AST semantics. We use episode mining to learn different API usage pattern types from source code, perform an empirical study to compare the different pattern types based on three pre-defined metrics (Section 4.4.3), build an API misuse detector based on the learned patterns, and compare how it performs with a state of the art misuse detector (MuDetect [13]), in terms of precision and recall. We make our work publicly available through PtBench benchmark, which consist of the following components: *(1)* Two data sets providing support for two different programming languages, C# and Java. *(2)* An adaptation of an event mining algorithm to the special context of pattern mining for software engineering. *(3)* Three well defined metrics, on which we base the empirical comparison between the different pattern types. *(4)* EmDetect to evaluate the effectiveness of the different pattern types within the application context of misuse detection.

Our empirical investigation shows that there are different types of patterns learned in code repositories. While there are tradeoffs between pattern types in terms of *expressiveness*, *consistency* and *generalizability*, they are comparable in terms of the patterns size and number of API types. Our results empirically show that the *sweet spot* in representing source code are *partial-order* patterns, which are a superset of *sequential-order* patterns, without losing valuable information like *no-order* patterns. Partial-order mining learns additional patterns compared to sequence mining, which generalize across repositories. In the application context of misuses detection, this results in better performance of partial-order patterns ($EMDetect_{POC}$) in terms of recall, but very low precision compared to sequential-order patterns ($EMDetect_{SOC}$). Compared to other detectors from the literature, ($EMDetect_{SOC}$) outperforms all of them in terms of precision, and ($EMDetect_{POC}$) performs the same (compared to JADET) or better (compared to three others) in terms of recall in the typical per-project setting. .

Our findings are useful indications for researchers who work with code patterns in applications of code recommendation and misuse detection.

In this chapter, we present a brief overview over the findings from this thesis. We start with a review of the results and contributions of this thesis and follow with a closing discussion.

6.1 Summary of Results

This thesis contributes to the area of code recommender systems. We provide a holistic view on the problem space and the state of the art in existing learning approaches from source code. We adapt matrix factorization and episode mining, two general-purpose machine learning algorithms within the application domains of code recommendation and misuse detection respectively, and compare their results with the latest state of the art approaches, PBN [123] and MUDETECT [13]. Finally, we present evidence for possible directions towards further improvement, as a stepping stone for future work.

Survey of State-of-the-art Pattern Mining Approaches We present a systematic literature review of 65 existing learning approaches from source code, based on both static and dynamic program analyses. From the findings of our literature review, we learn: (1) that approaches analyze a small subset of code elements, mostly neglecting elements other than method calls. (2) that 90% of the approaches are language dependent, because on relying on specific static and/or dynamic analysis. (3) that learning algorithms used to generate knowledge from source code are either based on simple frequency occurrences of code elements extracted, or require domain specific knowledge for feature extraction or natural language annotated code snippets. This shows the potential of using more sophisticated machine learning algorithms that can at the same time rely on code artifacts extracted using less complex program analyses, and automatically discover latent knowledge from source code which was not possible from existing approaches.

Improving scalability in existing code recommenders We adapted Boolean Matrix Factorization (BMF) within an existing code recommender pipeline based on canopy clustering. The reasons for using BMF over simple clustering algorithms are: (1) BMF can automatically calculate the number of clusters needed to represent a given object space, instead of inputing it as a user-defined parameter. (2) BMF is able to automatically identify noise from the data, while clustering algorithms partition any usage into some cluster. We compare BMF with canopy clustering and show that by maintaining the same recommendations quality, BMF is able to significantly improve model size and inference speed by up to 80%. In the experimental evaluation, BMF shows to be the best all-round performer, but this quality comes with a cost in running times. However, as each type is dealt independently, the work can be trivially distributed in the cloud, alleviating in this way the problem. Our results suggest that BMF is promising in the context of intelligent method call completion, and speculate that other software applications may also benefit from it.

Automated Benchmark for API-Usage Pattern Types We build PtBench, which to the best of our knowledge, is the first-ever automated empirical benchmark for comparing different API-pattern types (sequential, partial and no-order) with respect to real code. PtBench enables systematic, comparable, and reproducible experiments. It automates large parts of the evaluation process, including the transformation of source code into a simple stream of events (method declarations and invocations), the automatic calculation of frequency and entropy thresholds for the episode mining algorithm depending on the project size, the incorporation of an existing episode-mining algorithm for learning different source code representations, and comparison of the different pattern types according to three pre-defined metrics. Furthermore, PtBench is easily extensible by new data sets (programming languages), new metrics definitions, and additional SE application contexts for further experiments.

Our investigation shows that while there are tradeoffs between pattern types in terms of expressiveness, consistency and generalizability, they are comparable in terms of the patterns size and number of API types. Our empirical comparison showed practical evidence that partial-order patterns are a good trade-off for representing concrete code usages in the wild: they learn a high coverage of API usage patterns that are used across different repositories when compared to sequential-order patterns while ensuring that different variations of the same pattern are not redundantly stored, and at the same time maintain important order information when compared to no-order patterns. Evaluation results show that all three configurations end-up learning only repository-specific patterns for pattern sizes with $6 - events$ or more. Furthermore, our experimental results proved the importance of order information in sequential and partial-order patterns for representing source code in the wild. Our findings are useful indications for researchers working with code patterns/snippets/examples in applications such as code recommendation, misuse detection, code search etc.

The importance of order information in misuse detection We use PtBench for a systematic evaluation and comparison of sequential and partial-order patterns for the purpose of misuse detection in Java projects. For this, we developed EmDetect, a new API method call misuse detector. EmDetect employs a pattern-mining and a violation-detection algorithm that efficiently and effectively identifies usage patterns and misuses based on *episodes*.

We find that both pattern types may successfully identify many misuses but suffer from extremely low precision (below 15%) and recall (below 42.1%) in a practical setting. However, our empirical evaluations show that even though EmDetect is based on a simple AST parser for extracting code artifacts, the results are still comparable in terms of both precision and recall, when compared to other state of the art approaches that heavily rely on detailed domain knowledge program analyses.

6.2 Future Work

In this section, we present our ideas for future work, with respect to advancing the state of the art in discovering latent knowledge from source code. For some of these ideas we present preliminary results, and others we identify as interesting challenges.

Data Sampling Is more data better, or can we achieve the same results by sampling a small portion of relevant data instead? To increase the generalizability of their results, many current approaches aim to experiment with very large amount of data. This usually results in scalability issues in current systems, and lots of noise in the data used during the training process. Data sampling is used over decades by statisticians, in cases when analyzing all the possible available data is practically impossible. Future work might investigate techniques for the retrieval of high-quality *representative* usage examples instead, and compare their results with existing work. Given the tremendous increase of the availability of source code data, future work should develop approaches that combine data from multiple sources to increase generalizability of the results, and at the same time incorporate sampling techniques to maintain the SE systems scalable.

Data preprocessing Agrawal *et al.* [5] showed that better data have a dominant impact in improving the obtained results. To obtain better data, preprocessing steps are required in order to improve the quality of the data used in training the models. Our BMF approach presented in Chapter 3, automatically discovered a high quantity of outliers in the data (API type `Table`), and episode mining suffered from low rates of generalizability in the learned patterns because of including testing code (Section 4.6.4) while training the models. Furthermore, the code surrounding an API usage may introduce noise. Detailed program analyses generate a high amount of data, but the question is which of this data is actually important for a task at hand, and how can they be used to distinguish between high quality API usage examples and noise in the analyzed source code? Is any data source relevant, or should we be careful when choosing the training source based on some specific parameters? What kind of filters do we need to apply on the increased amount of available code data on the different sources? These are some of the questions that need to be considered before generating the training data set in current systems. While measuring code quality remains generally an open question, using simple indicators like project maturity, code churn, or number of tests might already improve the results. Future work should investigate respective possibilities and their impact on current software engineering applications.

Automatic feature selection/pruning Feature selection have significantly improved results in other fields, especially towards reducing the size of the trained models and improving scalability of systems in general. Also, feature selection techniques allow to analyze the impact of different artifacts in the quality of the results obtained in general. Many current software engineering approaches are based on extracting a large amount of features from source code, such as in code recommendation [123], bug detection [66], code synthesis [170] etc. Applying feature pruning techniques on these approaches, might results in improved scalability and performance.

The support of API patterns The most common way to determine the support of API patterns is by counting the number of usages that adhere to the pattern in the source code. It follows the intuition that a pattern that holds more frequently is more likely to be a valid pattern and that is used across repositories and developers. But this is not always the case, as shown in Section 4.6.4 only half of the patterns we mine are generalizable. A possible alternative is the *method support*, which counts the number of methods that contain at least one usage following the pattern. If we interpret methods as code units implementing a particular task, the method support can be interpreted as the number of tasks using a certain API according to the mined pattern. The method support is smaller than or equal to the occurrence support, as it ignores additional usages following the same pattern within the implementation of the same task. Also, multiple usages within the same task are likely written under the same conception of the APIs pattern, thus not contributing with additional information to the training data. We hypothesize that this might improve the generalizability of the mined API patterns across different tasks and developers.

Another alternative is the *project support*, which counts the number of projects that contain at least one usage following the pattern. If we interpret projects as the work of different development teams, the project support can be interpreted as a measure of how many teams believe this pattern to be correct. We hypothesize that this might be an even better indicator for correctness than occurrence or method support, since it shows the popularity of the mined pattern across different developer teams. Our results in Section 4.6.4 support this hypothesis since 98% of the patterns learned finds usability across methods, but less than 50% of them generalize across repositories (development teams).

Future work should investigate and compare the impact of these (and possible other) ways to calculate the support of patterns on their quality and generalizability. Moreover, future work may investigate ways to combine different support metrics.

Frequency threshold for pattern mining Most of the existing approaches in pattern mining are based on *absolute thresholds* to mine API patterns. This follows the intuition that a pattern that holds at least a certain number of times, is likely to be correct. However, larger projects tend to have a higher number of training examples compared to smaller projects, which usually results in learning more API patterns from the larger projects while many code examples from smaller projects do not reach this threshold and are hence ignored. We mitigate this point in Section 5.3.2, where we generate *relative thresholds* according to the project size (number of method declarations and invocations).

This follows the intuition that a correct specification should hold more often in a larger training dataset, while also considering other (correct) usages from smaller projects.

However, our statistics in Table 4.2 shows that the number of patterns learned decreases significantly while increasing the pattern size. This implies that larger patterns require smaller thresholds, since their occurrence in source code is much lower. Future work might consider several different thresholds at once, considering as well the pattern size. For example, Saied *et al.* [137] first mine specifications with a high absolute threshold and then successively mine extensions to these specifications using ever lower thresholds. This follows the observation that there is often a strict core specification that all usages of an API must adhere to, and several alternative extensions to it.

To the best of our knowledge, no previous work systematically compared alternative ways to determine frequency thresholds. Future work should investigate how they impact the quality of the learned patterns within specific software engineering application contexts.

The Impact of API Usages Throughout the Development Process Developers struggle with API usages at development time, and these usages might be different from the patterns we identify. However, we know little about the actual impact of API usages at different stages of the development process and the potentially distinctive properties of usages at any particular stage. To find out more, future work could mine intermediate commits, or conduct surveys and field studies to learn more about which problems developers face at different stages. This would enable a systematic comparison to reveal whether there are indeed differences and how these impact research on API-pattern learning. Furthermore, such findings would contribute in building better recommender systems to assist developers throughout different development stages.

Learning from additional code elements We learn code patterns only for method declarations and invocations, excluding all other code elements such as loops, conditions, exceptions etc. This is because the focus of this thesis is to investigate the impact of more sophisticated machine learning algorithms in automatically discovering latent knowledge from source code, instead of specifically learning complex patterns that include all code elements. Since learning code patterns while considering other code elements is important for supporting certain development tasks and identifying other types of misuses, we address this as future work. This requires modifying our event stream generation, which is an engineering task rather than a conceptual limitation. However, the new results might bring some very interesting facts which are worth investigating.

Parallelizing machine learning algorithms Both BMF and episode mining algorithms used in this thesis are available only in a sequential (non-parallelized) implementation, hence they are inefficient. However they do not impact the scalability of the recommender systems based on them, since the models are trained only once and then used for either listing recommendations or detecting misuses. Future work should consider parallelizing such general purpose machine learning algorithms' implementation in order to make them applicable also in cases when the training data is continuously updated, and as a consequence the models need to be regenerated multiple times.

Experimenting on Further Applications Based on API-Usages Our work is limited to two application contexts: code recommendation and misuse detection. Our survey in Section 2.4.3 identifies at least eight other software engineering applications, which realize quite distinctive ideas. We believe it is important to incorporate other applications, in order to get a more complete picture on the impact of general-purpose machine learning algorithms within the software engineering context, and identify the respective strengths and weaknesses. As the work presented in this thesis shows, such systematic assessment can reveal opportunities for significant improvements. However, future work needs to systematically investigate the effects and counter-effects of individual techniques, to see whether we can balance them out.

6.3 Closing Discussion

Reuse of existing software components is an integral ingredient to efficient software development. Software developers use such components through their APIs. Thereby, they must consider the usage constraints that come with these APIs.

Researchers have dedicated much work to the automated learning of API patterns. This thesis consolidates over a decade of research for a qualitative and quantitative assessment of the state of the art. We find that existing approaches conceptually cover only a small subset of all types of API elements. Moreover, we find that existing approaches often rely on language-dependent program analyses to extract domain-knowledge code artifacts from software, and then apply simple mining algorithms to aggregate the extracted information.

In this thesis, we employ two state of the art general purpose machine learning algorithms in order to automatically extract latent knowledge from source code that was not possible with previous approaches: (1) *Boolean Matrix Factorization* (BMF) automatically calculates the number of clusters needed to represent a given object space, and at the same time identifies noise in the data. (2) *Episode mining* uses a simple representation of source code (stream of events) to automatically identify different code structures (pattern types): sequential, partial, and no-order. Furthermore, we design a new static API-misuse detector (EmDetect) to be able to compare how the different pattern types generated from the episode mining algorithm, compare within the application domain of misuse detection. EmDetect achieves a precision up to 15% and a recall up to 42.1%, which is comparable to other existing approaches from the literature.

The work presented in this thesis shows that there is potential for further improvement in existing software engineering applications, for automatically discovering valuable latent knowledge from source code. For this purpose, the community needs to adapt more sophisticated general-purpose machine learning algorithms, while developing software engineering applications. We present several challenges that future work should address, in order to reach this goal.

First, current approaches employ rather simple learning algorithms, that are mainly frequency-based, i.e., they assume that frequent usages are correct and infrequent usages, consequently, incorrect. These learning algorithms focus on simply aggregating the extracted code artifacts instead of new information discovery. We should search for alternative learning algorithms to replace pure frequency, and develop (adapt) techniques that are able to generate latent knowledge from the available data.

Second, our experiments show that current general purpose machine learning algorithms are severely limited in scalability. Both approaches used in this thesis, BMF and episode mining, do not scale to larger training data sets, widely used in software engineering approaches. Parallelizing such algorithms will significantly improve their scalability and applicability in software engineering and other domains. At the same time, we should also investigate the quality of the training examples we use in such approaches.

Contributed Implementations and Data

In the course of the projects presented in this thesis, research prototypes have been implemented and data has been collected. We provide these implementations and datasets to enable other researchers to validate our work and to build new research upon them. We believe this to be good scientific practice and encourage other researchers to do the same.

Boolean Matrix Factorization (BMF)

For the work presented in Chapter 3, we provide in the following link the corresponding artifact page, containing information regarding the data set used to perform the experiments, the BMF algorithm implementation used for mining the patterns, the complete pipeline source code implementation, and other useful information for replicating the experiments:

`http://www.st.informatik.tu-darmstadt.de/artifacts/bmf4cr/`

The BMF algorithm used is fully integrated into our implementation pipeline in Java (BMF4CR), to allow continuous execution in the context of the respective experiments. Our BMF4CR project uses APACHE MAVEN for its build in configuration, to allow system- and IDE-independent builds.

Pattern Types Benchmark (PTBench)

PTBENCH is the benchmarking pipeline introduced in Chapter 4 for comparing different pattern types based on three predefined metrics, and it's performance is evaluated within the application context of misuse detection presented in Chapter 5 where we introduce EMDETECT. We provide the implementation of EMDETECT presented in Section 4.3, which is fully integrated into PTBENCH to avoid interruption in executing the respective experiments. We make publicly available the data set, the entire source code implementation, the mined patterns and other useful information for replicating the experiments on the following artifact page:

`http://www.st.informatik.tu-darmstadt.de/artifacts/patternTypes/`

The EMDETECT project uses APACHE MAVEN for its build configuration, to allow system- and IDE-independent builds. The detector can be built using the `mvn package` command, which creates standalone bundles called `EmDetect.jar` for the detector in the `target` folder. The `README` files of the GitHub repository present further details on the organization of the project and its code.

On the artifact page, we provide as well the review results and additional data artifacts for the evaluations presented in Section 5.4. The full experiment data may be downloaded from the review sites in CSV format or viewed online.

Bibliography

[1] A. Achar, S. Laxman, R. Viswanathan, and P. S. Sastry. Discovering injective episodes with general partial orders. *Data Mining and Knowledge Discovery*, 25(1):67–108, Jul 2012.

[2] A. Achar and P. Sastry. Statistical significance of episodes with general partial orders. *Information Sciences*, 296:175–200, 2015.

[3] M. Acharya and T. Xie. Mining api error-handling specifications from source code. In *International Conference on Fundamental Approaches to Software Engineering*, pages 370–384. Springer, 2009.

[4] 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*, pages 25–34, 2007.

[5] A. Agrawal and T. Menzies. Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1050–1061. ACM, 2018.

[6] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, 1993.

[7] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*, pages 3–14. IEEE, 1995.

[8] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.

[9] G. Alonso, C. Hagen, D. Agrawal, A. El Abbadi, and C. Mohan. Enhancing the fault tolerance of workflow management systems. *IEEE Concurrency*, 8(3):74–81, 2000.

[10] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109, 2005.

[11] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13^{th} Working Conference on Mining Software Repositories*, MSR '16. ACM Press, 2016.

[12] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. A systematic evaluation of static API-misuse detectors. *IEEE Transactions on Software Engineering*, 2018.

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

[14] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

[15] M. W. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52:155–173, 2007.

[16] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, 100(6):592–597, 1972.

[17] E. Bodden, P. Lam, and L. Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):7, 2012.

[18] B. Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.

[19] N. P. Borges, M. Gómez, and A. Zeller. Guiding app testing with mined interaction models. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 133–143. IEEE, 2018.

[20] H. B. Braiek, F. Khomh, and B. Adams. The open-closed principle of modern machine learning frameworks. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 353–363. IEEE, 2018.

[21] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[22] M. Bruch, T. Schäfer, and M. Mezini. Fruit: Ide support for framework understanding. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, pages 55–59, New York, NY, USA, 2006. ACM.

[23] 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*, pages 242–251. ACM, 2006.

[24] 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.

[25] R. P. L. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA'08. ACM Press, 2008.

[26] T. Cargill. Exception handling: A false sense of security. In *C++ gems*, pages 423–431. SIGS Publications, Inc., 1996.

[27] E. Çergani and M. Mezini. On the impact of order information in api usage patterns. In *International Conference on Software Technologies*, pages 79–103. Springer, 2018.

[28] E. Cergani and P. Miettinen. Discovering relations using matrix factorization methods. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 1549–1552, New York, NY, USA, 2013. ACM.

[29] E. Cergani, S. Proksch, S. Nadi, and M. Mezini. Addressing scalability in api method call analytics. In *Proceedings of the 2Nd International Workshop on Software Analytics*, SWAN 2016, pages 1–7, New York, NY, USA, 2016. ACM.

[30] E. Cergani, S. Proksch, S. Nadi, and M. Mezini. Investigating order information in api-usage patterns: A benchmark and empirical study. In *ICSOFT*, pages 91–102, 2018.

[31] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of points-to analysis of java in the presence of exceptions. *IEEE Transactions on Software Engineering*, 27(6):481–512, 2001.

[32] C. Cheng, H. Yang, I. King, and M. R. Lyu. Fused matrix factorization with geographical and social influence in location-based social networks. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[33] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 21–31. ACM, 1999.

[34] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96. ACM Press, 2010.

[35] C. De Roover, R. Lammel, and E. Pek. Multi-dimensional exploration of api usage. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 152–161. IEEE, 2013.

[36] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330. IEEE Computer Society Press, 2009.

[37] M. Dias, D. Cassou, and S. Ducasse. Representing code history with development environment events. *arXiv preprint arXiv:1309.4334*, 2013.

[38] T.-A. Doan, D. Lo, S. Maoz, and S.-C. Khoo. LM: A miner for scenario-based specifications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2 of *ICSE '10*, pages 319–320. ACM Press, 2010.

[39] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349, 2008.

[40] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24. ACM Press, 2010.

[41] B. Ganter and R. Wille. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

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

[43] P. D. Grünwald and A. Grunwald. *The minimum description length principle*. MIT press, 2007.

[44] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISTA '10, pages 119–129. ACM Press, 2010.

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

[46] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[47] D. Guillamet and J. Vitria. Non-negative matrix factorization for face recognition. In *Catalonian Conference on Artificial Intelligence*, pages 336–344. Springer, 2002.

[48] J. Haase and U. Brefeld. Mining positional data streams. In *International workshop on new frontiers in mining complex patterns*, pages 102–116. Springer, 2014.

[49] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[50] A. E. Hassan. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57. IEEE, 2008.

[51] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE 2012)(ICSE)*, pages 837–847, June 2012.

[52] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125. IEEE Computer Society Press, 2005.

[53] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[54] G. Hu, L. Zhu, and J. Yang. Appflow: using machine learning to synthesize robust, reusable ui tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 269–282, 2018.

[55] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM, 2018.

[56] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[57] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, pages 20–30, 2018.

[58] C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *European Conference on Object-Oriented Programming*, pages 27–51. Springer, 2009.

[59] M. Kersten and G. C. Murphy. Mylar: A degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM.

[60] P. Koehn. Europarl: A Parallel Corpus for Statistical Machine Translation. In *Conference Proceedings: the tenth Machine Translation Summit*, pages 79–86, Phuket, Thailand, 2005. AAMT, AAMT.

[61] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[62] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 2019.

[63] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 591–600. ACM, 2011.

[64] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321, 2011.

[65] X. Li, H. Jiang, D. Liu, Z. Ren, and G. Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension*, pages 144–155. ACM, 2018.

[66] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315, 2005.

[67] C. Lindig. Mining patterns and violations using concept analysis. In *The Art and Science of Analyzing Software Data*, pages 17–38. Elsevier, 2015.

[68] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*, pages 681–690, 2010.

[69] C. Liu, E. Ye, and D. J. Richardson. Software library usage pattern extraction using asoftware model checker. *International Journal of Computers and Applications*, 31(4):247–259, 2009.

[70] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 2018.

[71] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305, 2005.

[72] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[73] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Rosu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Runtime Verification*, pages 285–300. Springer-Verlag GmbH, 2014.

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

[75] H. Ma, R. Amor, and E. D. Tempero. Usage patterns of the java standard api. *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 342–352, 2006.

[76] Y. Ma, S. Fakhoury, M. Christensen, V. Arnaoudova, W. Zogaan, and M. Mirakhorli. Automatic classification of software artifacts in open-source applications. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 414–425. IEEE, 2018.

[77] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *ACM Sigplan Notices*, volume 40, pages 48–61. ACM, 2005.

[78] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, pages 259–289, 1997.

[79] R. C. Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall PTR, 2003.

[80] F. Mccarey, M. Ó. Cinnéide, and N. Kushmerick. Rascal: A recommender agent for agile reuse. *Artificial Intelligence Review*, 24(3-4):253–276, 2005.

[81] D. Mendez, B. Baudry, and M. Monperrus. Empirical evidence of large-scale diversity in API usage of object-oriented software. In *Source Code Analysis and Manipulation*, pages 43–52, 2013.

[82] X. Meng and B. P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[83] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 554–563. IEEE, 2018.

[84] Y. Merhav, F. Mesquita, D. Barbosa, W. G. Yee, and O. Frieder. Extracting information networks from the blogosphere. *ACM Trans. Web*, 6(3):11:1–11:33, Oct. 2012.

[85] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 167–176, New York, NY, USA, 2000. ACM.

[86] P. Miettinen. On the positive–negative partial set cover problem. *Information Processing Letters*, 108(4):219–221, 2008.

[87] P. Miettinen. *Matrix decomposition methods for data mining: Computational complexity and algorithms.* PhD thesis, Helsingin yliopisto, 2009.

[88] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila. The discrete basis problem. *IEEE transactions on knowledge and data engineering*, 20(10):1348–1362, 2008.

[89] P. Miettinen and J. Vreeken. Model order selection for boolean matrix factorization. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 51–59, 2011.

[90] P. Miettinen and J. Vreeken. Mdl4bmf: Minimum description length for boolean matrix factorization. *ACM Trans. Knowl. Discov. Data*, 8(4):18:1–18:31, Oct. 2014.

[91] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP '10, pages 2–25. Springer-Verlag GmbH, 2010.

[92] M. Monperrus and M. Mezini. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology*, 22:1–25, 2013.

[93] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. Documenting apis with examples: Lessons learned with the apiminer platform. In *WCRE*, pages 401–408. IEEE Computer Society, 2013.

[94] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 13–, Washington, DC, USA, 2001. IEEE Computer Society.

[95] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890, Piscataway, NJ, USA, 2015. IEEE Press.

[96] V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 151–162, 2017.

[97] G. C. Murphy, R. J. Walker, and R. Holmes. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32:952–970, 2006.

[98] N. Nachar. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1):13–20, mar 2008.

[99] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. *ACM Sigplan Notices*, 43(10):347–366, 2008.

[100] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, pages 803–813, 2014.

[101] A. C. Nguyen and S.-C. Khoo. Extracting significant specifications from mining through mutation testing. In *Formal Methods and Software Engineering*, pages 472–488. Springer-Verlag GmbH, 2011.

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

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

[104] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468, 2014.

[105] 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. IEEE Computer Society Press, 2015.

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

[107] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in javascript web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 791–802, 2014.

[108] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta. Focus: A recommender system for mining api function calls and usage patterns. In

2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 1050–1060. IEEE, 2019.

[109] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 315–324, 2010.

[110] 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 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pages 383–392, 2009.

[111] 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. IEEE, 2015.

[112] 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. ACM Press, 2016.

[113] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 815–825. IEEE, 2012.

[114] V. P. Pauca, F. Shahnaz, M. W. Berry, and R. J. Plemmons. Text mining using non-negative matrix factorizations. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 452–456. SIAM, 2004.

[115] J. R. Peddamail, Z. Yao, Z. Wang, and H. Sun. A comprehensive study of staqc for deep code summarization. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2018.

[116] L. Ponzanelli, A. Mocci, and M. Lanza. Stormed: Stack overflow ready made data. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 474–477, Piscataway, NJ, USA, 2015. IEEE Press.

[117] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[118] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 371–382. IEEE, 2009.

[119] M. Pradel and T. R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 232–242. ACM, 2011.

[120] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 288–298. IEEE, 2012.

[121] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 925–935. IEEE, 2012.

[122] S. Proksch, S. Amann, S. Nadi, and M. Mezini. A dataset of simplified syntax trees for c#. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 476–479, 2016.

[123] S. Proksch, J. Lerch, and M. Mezini. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25:1–31, 2015.

[124] D. Qiu, B. Li, and H. Leung. Understanding the api usage in java. *Information and software technology*, 73:81–100, 2016.

[125] J. Quante and R. Koschke. Dynamic protocol recovery. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 219–228. IEEE, 2007.

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

[127] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proceedings of the 28^{th} ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 123–134. ACM Press, 2007.

[128] T. Ramraj and R. Prabhakar. Frequent subgraph mining algorithms – a survey. *Procedia Computer Science*, 47:197–204, 2015.

[129] 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*, pages 419–428, 2014.

[130] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[131] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE software*, 27(4):80–86, 2009.

[132] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39:613–637, 2013.

[133] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer-Verlag GmbH, 2014.

[134] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003.

[135] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman. Labelme: A database and web-based tool for image annotation. *International Journal of Computer Vision*, 77(1):157–173, May 2008.

[136] I. Saenko and I. V. Kotenko. Design of virtual local area network scheme based on genetic optimization and visual analysis. *JoWUA*, 5(4):86–102, 2014.

[137] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level api usage patterns. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 23–32. IEEE, 2015.

[138] M. A. Saied and H. Sahraoui. A cooperative approach for combining client-based and library-based api usage pattern mining. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[139] M. A. Saied, H. Sahraoui, and B. Dufour. An observational study on api usage constraints and their documentation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 33–42. IEEE, 2015.

[140] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 155–164. IEEE, 2005.

[141] K. Schmid. Top productivity through software reuse. In *12th International Conference on Software Reuse, Lecture Notes in Computer Science, ICSR, Springer*, volume 6727. Springer, 2011.

[142] W. Schwittek and S. Eicker. A study on third party component reuse in java enterprise open source software. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pages 75–80, 2013.

[143] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[144] V. Singh, L. L. Pollock, W. Snipes, and N. A. Kraft. A case study of program comprehension effort and technical debt estimations. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–9. IEEE, 2016.

[145] V. Snael, P. Kromer, J. Platos, and D. Husek. On the implementation of boolean matrix factorization. In *Proceedings - International Workshop on Database and Expert Systems Applications, DEXA*, pages 554 – 558, 10 2008.

[146] 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.

[147] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, 2007.

[148] S. Thummalapenta and T. Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336. IEEE, 2008.

[149] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294. IEEE, 2009.

[150] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31^{st} International Conference on Software Engineering*, ICSE '09, pages 496–506. IEEE Computer Society Press, 2009.

[151] F. Tip. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[152] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis for software maintenance. In *Proceedings. First Euromicro Conference on Software Maintenance and Reengineering*, pages 60–67. IEEE, 1997.

[153] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553. IEEE, 2018.

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

[155] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings. 20th international conference on data engineering*, pages 79–90. IEEE, 2004.

[156] X. Wang, L. Pollock, and K. Vijay-Shanker. Developing a model of loop actions by mining loop characteristics from a large code corpus. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 51–60. IEEE, 2015.

[157] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[158] 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*, pages 35–44, 2007.

[159] M. Weimer, A. Karatzoglou, and M. Bruch. Maximum margin matrix factorization for code recommendation. In *Proceedings of the third ACM conference on Recommender systems*, pages 309–312, 2009.

[160] 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*, pages 419–431, 2004.

[161] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[162] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[163] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 218–228. ACM, 2002.

[164] 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*, pages 334–345. IEEE Press, 2015.

[165] W. Xu, X. Liu, and Y. Gong. Document clustering based on non-negative matrix factorization. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 267–273, 2003.

[166] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, pages 282–291. ACM, 2006.

[167] J. Yang, E. Wittern, A. T. Ying, J. Dolby, and L. Tan. Towards extracting web api specifications from documentation. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 454–464. IEEE, 2018.

[168] Z. Yao, D. S. Weld, W.-P. Chen, and H. Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703, 2018.

[169] Y. Ye and G. Fischer. Reuse-conducive development environments. *Automated Software Engineering*, 12(2):199–235, 2005.

[170] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE, 2018.

[171] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[172] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 826–836. IEEE, 2012.

[173] D. Zhang, Y. Guo, and X. Chen. Automated aspect recommendation through clustering-based fan-in analysis. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 278–287. IEEE, 2008.

[174] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 886–896. IEEE, 2018.

[175] Y. Zhang, D. Lo, X. Xia, J. Jiang, and J. Sun. Recommending frequently encountered bugs. In *International Conference on Program Comprehension, Gothenburg, Sweden*, 2018.

[176] H. Zhong and H. Mei. An empirical study on api usages. *IEEE Transactions on Software Engineering*, 45(4):319–334, 2017.

[177] H. Zhong and Z. Su. Detecting API documentation errors. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages & Applications*, volume 48, pages 803–816. ACM Press, 2013.

[178] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[179] H. Zhong, L. Zhang, and H. Mei. Inferring specifications of object oriented apis from api source code. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 221–228. IEEE, 2008.

[180] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318. IEEE, 2009.

[181] C. Zhou, B. Li, X. Sun, and H. Guo. Recognizing software bug-specific named entity in software bug repository. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 108–119. ACM, 2018.

[182] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.