5-2018

Recommending APIs for software evolution

Ferdian THUNG
Singapore Management University, ferdiant.2013@phdis.smu.edu.sg

Recommending APIs for Software Evolution

FERDIAN THUNG

SINGAPORE MANAGEMENT UNIVERSITY

2018

Recommending APIs for Software Evolution

by

Ferdian Thung

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Singapore Management University
2018

Recommending APIs for Software Evolution

Ferdian Thung

Abstract

Softwares are constantly evolving. This evolution has been made easier through the use of Application Programming Interfaces (APIs). By leveraging APIs, developers reuse previously implemented functionalities and concentrate on writing new codes. These APIs may originate from either third parties or internally from other components of the software that are currently developed. In the first case, developers need to know how to find and use third party APIs. In the second case, developers need to be aware of internal APIs in their own software. In either case, there is often too much information to digest. For instance, finding the right APIs may require sifting through many different APIs and learning them one by one, which can easily cost a large amount of time. Also, as the software becomes bigger and more complex, developers may not be aware of all functionalities available in their software.

To deal with the above-mentioned difficulties, we propose API recommendation approaches for software evolution. We have developed four approaches in this direction. The first three approaches assist developers in using third party APIs while the fourth approach assists developers in evolving software according to changes in its internal APIs. Our first approach deals with a problem of finding the right API libraries for implementing new software features. Given a list of current libraries used in a software, our approach combines association rule mining and collaborative filtering to recommend libraries that are potentially useful. Our fourth work deals with the same problem as our first work. We improve upon our first work on automatic API recommendation by considering API library description and adding matrix factorization in the mix. Description of each API library will be used to improve one of the component of our previous approach while matrix factorization will be used to create a new recommendation component. We combine the updated components

of our previous approach and our new recommendation component to build a recommendation system that is better than our previous approach. Our third approach deals with the subsequent problem after finding the right API libraries, which is finding the right API methods to be used for implementing the new software features. Given a description of feature to be implemented and known libraries, our approach combines historical and descriptive information to recommend API methods in the known libraries that can be used to implement the feature.

Our fourth work deals with a problem of evolving an older software version to contain features and/or bug fixes that are added in the newer software version. This is necessary since some softwares still maintain their older versions. One of the biggest example is Linux kernel. New device drivers are consistently added and/or updated in the latest kernel version. However, many systems still use older kernel versions and thus these new device drivers also need to be available in the older versions. Given a code in the new version that implements a new feature or bug fix, our approach can recommend how the code can be changed to work in the old version. Our approach does so by finding a portion of the code that causes an error when the code is directly used in the old version. Our approach then search the history for suggestions on how to transform the code to its equivalent form that can work with the other pieces of code in the old version. Our approach recommend this equivalent form in order to make the new code works in the old version and thus making the new feature or bug fix available in the old version.

Overall, this dissertation aims to assist developers by providing useful recommendations that they can use to help them in evolving their software. Experiments to evaluate our approaches have shown that they can perform accurate recommendations.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to the people that have supported me throughout the course of my PhD journey.

First, I would like to thank my supervisor, Prof. David Lo, for his continuous support and guidance that have helped me in growing my research skills and experiences.

I would like to say thank you to members of SOftware Analytics Research (SOAR) group, and all co-authors that I have worked with for their supports in our collaborative work.

I would also like to thank my dissertation committee: Prof. Lingxiao Jiang, Prof. Hady W. Lauw, and Dr. Julia Lawall for their comments and feedbacks that have improved my dissertation.

Last but not least, I would like to thank my family for their constant and endearing support throughout my studies.

Chapter 1

Introduction

This chapter discusses the main problems and motivations of this dissertation. A summary of works completed and the structure of this dissertation are then presented.

1.1 Motivations

Gone are the days when developers have to build a software from scratch. The proliferation of third party libraries makes software development task faster and easier than ever before. In fact, libraries have been an integral part of software development [7, 30, 64]. Indeed, using third party libraries allows developers to save time since they do not need to reinvent the wheel. Instead, they can focus on the specific task at hand. Third party libraries are also tested; thus, they reduce the likelihood of creating bugs.

Although third party libraries are readily available, finding relevant libraries and using them are not always straightforward. Developers need to find relevant libraries among thousands and manually check whether the library functionality is a match with their requirements. This manual checking might involve a long process involving finding relevant methods in the API and understanding how to use them. Comprehending the structure of an API and how to choose method parameters properly are among the things that developers need to learn. Clearly, this process can be time consuming and the existence of an API recommendation system would help developers a great deal in doing their job.

The above difficulties also exist when using internal APIs in the software that developers currently working on. As the software becomes more complex, developers may not know all the APIs available internally. They might then either reimplement the functionalities themselves or use external libraries to provide them with the functionalities they require. This might even happen when the functionalities are already available internally and thus the developers should have used these rather than opting for third parties or their own implementation.

When evolving a software, developers also need to be aware of changes in APIs. They must know how to provide the same functionalities in one version of the API versus the others. This scenario generally happens when there is a need to provide the same functionalities in different versions of the software. In this case, developers need to know how to perform changes in API uses so that their code would work in different API versions. This is not easy since developers may only know a specific version of APIs, e.g. the latest version. Different versions might have differences in data structures and method definitions.

The above problems might be encountered when evolving software and highlight the reasons why API recommendations would be beneficial. A good recommendation would remove the need for developers to manually search different kinds of APIs. It can directly present developers with possible APIs that they can use to implement the required task. Consequently, it saves developers time and speeds up the software evolution process in general.

1.2 Overview

We have completed four distinct works for this dissertation. The first three works deal with the problem of recommending usage of third party APIs while the fourth work deals with the problem of recommending code transformations to convert a piece of code from a particular API version to another API version. These transformations recommend how APIs should be changed between the two versions.

1.2.1 Automated Library Recommendation

In this work, we tackle the problem of recommending API libraries to developers. Given a list of currently used libraries, we want to recommend additional libraries that developers can leverage to develop their software. Our proposed library recommendation, named as $LibRec$, consists of two main components: $LibRec^{RULE}$ and $LibRec^{COLLAB}$. $LibRec^{RULE}$ makes use of library usage patterns to recommend libraries. The patterns are mined using association rule mining techniques. $LibRec^{COLLAB}$ makes use a nearest-neighbor-based collaborative filtering approach to recommend libraries that are used by similar projects.

In the training phase, our system extracts rules and feature vectors from a set of projects that contains information of third-party libraries that they use. Given the association rules and feature vectors, in the deployment phase, our system can recommend libraries to a set of new projects based on the third-party libraries that they already use. The two models would give a score for each library in the database. Our approach then computes a final recommendation score by linearly combining the two scores. Libraries having the highest scores will be recommended.

We have performed experiments on 500 software projects. These projects are taken from GitHub, written in Java, relatively large (i.e., contain more that 10,000 lines of code), are not a fork of another project, and already use at least ten third party libraries. Our experiments show that our approach can achieve recall-rate@5 and recall-rate@10 of 0.852 and 0.894, respectively. Recall-rate@5 means that there is at least one correct recommendation in the top-5 recommended libraries. This result shows the promise of our approach.

1.2.2 Improving Library Recommendation using Textual Content and Matrix Factorization

In this work, we tackle the problem of library recommendation. This work is the continuation of our first work. We improve upon our previous library recommendation approach by considering textual information from the library and adding a matrix factorization technique. This approach is built directly on top of our previous approach. Components of our previous approach are either left untouched or modified to cater for the newly considered information. We follow the same approach for the association rule mining based component. For the collaborative filtering based component, we employ a more fine grained representation of a software project by leveraging textual content of its adopted third party libraries. We then introduce a matrix factorization based component and combine it with the above components to generate a better library recommendation.

For the evaluation, we simulate the actual history of the library usage. When simulating a library recommendation, we use the initial libraries as input to our approach to evaluate whether it can recommend libraries that were added later on. Our experiment shows the effectiveness of our approach, achieving Hit@5 of 0.607 and Hit@10 of 0.702. Since our previous approach was evaluated differently, we rerun it in this setting. It achieves Hit@5 of 0.504 and Hit@10 of 0.640. Thus, our approach improves over our previous one by 20.44% and 9.69%, in terms of Hit@5 and Hit@10, respectively.

1.2.3 Automatic Recommendation of API Methods from Feature Requests

In this work, we tackle the problem of recommending API methods to developers. Given a task description and a list of known libraries, our approach can recommend API methods from the known libraries that can be used to implement the task.

Our proposed API method recommendation contains three major components:

a *History Based Recommender*, a *Description Based Recommender*, and an *Integrator*. The *History Based Recommender* takes as input the description of a new feature request, i.e. a task, and a historical database containing past feature requests. The recommender finds feature requests in the historical database that are the closest to the new feature request. The methods that are used to implement the closest feature requests are ranked higher. This recommendation is based on the collaborative filtering approach. The *Description Based Recommender* takes as input the description of the new feature request and the API libraries documentation. The recommender computes the similarity of the feature request description with each method description in the API documentation. Methods whose textual descriptions are the most similar with the new feature request description are more recommended. This recommendation is based on the information retrieval approach. The *History Based Recommender* and the *Description Based Recommender* produce recommendation scores for each API method. Our approach linearly combines the recommendation scores and returns a list of API methods ranked by the final combination score. The API method with the highest score is first recommended to the developer.

We have evaluated our approach on feature requests of 5 software projects and recommend methods from 10 libraries. We show that our approach achieves a recall-rate@5 and recall-rate@10 of 0.690 and 0.779, respectively.

1.2.4 Recommending Code Changes for Automatic Backporting of Linux Device Drivers

In this work, we tackle the problem of recommending code changes required to make code from the new version of a software project work in an old version of the software project. Such changes are required to make the features or bug fixes in a new version available in an older version. We call this process backporting and one setting in which it is relevant is Linux device drivers. New device drivers are first written to target the newest Linux kernel. However, many running systems use an older Linux kernel to maintain system stability so these device drivers are often backported to work in the older Linux kernel.

Our recommendation approach works by taking a new device driver implementation and the Linux kernel repository. It then finds the latest version in the repository where the new code induces a compilation error. The error information is then used to find hints on how to make the new code compilable in the old Linux kernel version. We also observe that the latest version in which the error occurs often contains examples on how the device driver should be transformed. We generate transformation recommendations based on these hints and rank the transformations by the likelihood of being correct.

We tested our approach on 100 Linux device drivers. We simulated backporting by pretending the device drivers do not exist in the old version. We then input the device drivers to our approach to receive transformation recommendations. We consider that a recommendation is correct if and only if the resulting transformed code exactly matches the corresponding device driver in the old version. Our experiments show that our approach can generate correct recommendations for 68% of the device drivers. Among these device drivers, 73.5% of the time our approach puts the correct recommendation as the first recommendation. We cannot generate correct recommendation for 32% of the device drivers because our search cannot find any relevant example or our recommendation can only provide a partial fix.

1.2.5 Structure of This Dissertation

Chapter 2 reviews previous works in literature. Chapter 3 describes *LibRec* – an API library recommendation approach. Chapter 4 describes *LibXplore*, an extension of *LibRec*. Chapter 5 describes an API method recommendation approach, followed by Chapter 6 which describes an approach that recommends how to backport Linux device drivers. Chapter 7 concludes this dissertation and presents some future work.

Chapter 2

Literature Review

We now present related work on code recommendation, identifying and analyzing changes, and mining change rules. We also present related work on recommendation systems in software engineering.

Code Recommendation. Mandelin et al. propose the tool Prospector, which recommends objects and method calls, referred to as jungloids, to convert an object of a particular type to an object of another type [44]. Prospector takes as input a query consisting of a pair of the input type and the output type. It then analyzes signatures of API methods and constructs a signature graph to recommend jungloids based on the query. A jungloid is ranked based on the number of methods it contains and the output type. Thummalapenta and Xie investigate the same problem [75]. However, they make use of a code search engine to solve the problem. The code search engine collects code examples which are then analyzed to recover the method sequences. While the approach by Mandelin et al. analyzes library code, the approach by Thummalapenta et al. analyzes client code retrieved by code search engines. Our work differs in several respects: we consider a different problem (method recommendation given a feature request vs. method recommendation given an input-output type pair), and we leverage different resources (historical feature requests + API documentation vs. library code or client code returned by a code search engine).

Bruch et al. propose a code completion system that recommends method calls by looking for code snippets in existing code repositories that share a similar context as the context that a developer is working on [11]. They propose three code completion systems based on frequency of method call usage, an association rule mining algorithm, and a k-nearest neighbor algorithm. The k-nearest neighbor algorithm performs the best. Our work differs in several respects: we consider different problems (method recommendation given a feature request vs. method recommendation given a code context), and we leverage different resources (historical feature requests + API documentation vs. code repositories).

Robillard proposes a technique, Suade, that recommends methods or other program elements of interest to help developers perform a software maintenance task [62]. Suade takes as input a set of program elements, and outputs other program elements that potentially interest developers. It works by investigating the structural dependencies of program elements and considers two criteria: specificity and reinforcement. A program element of interest needs to be specific enough to the input set and its relationship to the input set is reinforced by existing relationships among program elements in the input set. Saul et al. addresses a similar problem [67]. Their goal is to recommend a set of methods that are related to a target method. To achieve this goal, structural information in a call graph is analyzed. They propose a new algorithm named FRAN which performs random walk on the callgraph. Several other approaches recommend methods related to a target method using static analysis [43, 89]. Long et al. propose Altair that recommends method based on variables that are shared among related methods [43]. Zhang et al. enhance a call graph with control flow information and use it for method recommendation [89].

The closest work to ours is that of Chan et al., which recommends API methods given textual phrases [13]. Given a query expressed as a set of textual phrases, their approach returns a connected API subgraph. For this, they create an API graph, which is an undirected graph with nodes corresponding to classes and methods and edges corresponding to relationships between them (e.g., inheritance, input, output, and membership). Each node of the graph contains words that appear in the corresponding method. Based on this API graph and an input query, they mine a subgraph that maximizes a particular objective function. The approach is evaluated on a small number of short text phrases.

Identifying and Analyzing Changes. Many studies focus on identifying and analyzing changes between two code versions. For example, Neamtiu et al. [51] use partial abstract syntax tree matching to compare the source code of different C program versions. Horwitz et al. [26] focus on identifying semantic changes in programs. Fluri et al. [18] track the co-evolution of comments and source code over multiple program versions, leveraging *Evolizer* [20] and *ChangeDistiller* [19] for extracting fine-grained source code changes between program versions. Marinescu et al. [46] present *Covrig*, an infrastructure that collects static and dynamic software metrics when running different program versions, to study the co-evolution of source code and test cases. Zaidman et al. [87] examine the co-evolution of source code and tests, inferring the development style employed by a number of Java projects.

Mining Change Rules. Wu et al. [83] propose an approach to mine method call change rules between two versions of a Java program. They combine call dependency and text similarity analyses to extract method change rules. Similarly, Meng et al. [49] extract method change rules between two versions of a Java program. However, rather that considering only one big change between two versions of a program, they consider all changes between two consecutive commits that are located in the change history between the two versions of the program. This allows them to analyze changes in finer detail and enables them to mine chains of method changes. These works deal with a forward porting problem, which involves porting function calls to use a newer version of a library in Java programming language. We address a backporting problem, which is the task of porting a new version of some code (in our case, drivers) to work with an older version of a system (in our case, the Linux kernel). Thus, backporting can be seen as reverse forward porting. However, different than existing forward porting approaches, we have found that it is not sufficient to focus only on function calls, as we found the need to also port data structures.

Negara et al. [52] develop a tool for automatically mining *frequent* change rules from fine-grained edits. Different from their work, we focus on finding transformations that can backport drivers and these transformations are often not the frequent ones. Andersen et al. [3, 4] infer safe and concise transformation rules from a collection of transformation examples.

Recommendation Systems in Software Engineering. Robillard and Chhetri proposed an approach that recommends a portion of API documentation that is relevant to a given API element [63]. They categorized whether the portion is indispensable, valuable, or neither. McMillan et al. proposed Exemplar that searches relevant API method calls to recommend relevant applications [48]. Zhang et al. proposed Precise, a tool for recommending correct API parameters [88]. Precise constructs abstract parameter usage patterns by mining existing software projects. It recommends parameter values by analyzing development context, querying abstract parameters, and then concretizing them.

Anvik et al. applied a supervised machine learning algorithm on bug reports to recommend suitable developers that are likely capable to fix the described bugs in the reports [5]. Xia et al. proposed an approach that recommends developers which should be assigned to fix a bug report by combining bug report based analysis and developer based analysis [84]. Tian et al. proposed to recommend developers for fixing issues in bug reports by employing learning to rank approach [79]. Yang et al. proposed an approach to recommend similar bug reports by combining word embedding with a traditional information retrieval approach [85]. Ye et al. proposed to recommend source code files to consider when addressing a bug report using a learning to rank approach and similarity functions that they defined [86]. Source code files are then ordered based on their likelihood to be the source of the bug defined in the bug report. Almhana et al. proposed to use a multi objective optimization algorithm for recommending source code files to a bug report [2]. They defined two objective functions; the first one maximizes both lexical and historical similarities, while the second one minimizes the number of recommended classes.

Ponzanelli et al. proposed CodeTube, a tool that recommends portions of video tutorials that are related to a given query [56]. It combines image and textual analysis to recommend the related portions and provides related links in a Q&A programming website (i.e., StackOverflow). Rahman et al. proposed an approach that recommends relevant information when developers are facing programming errors and exceptions [59]. They exploited web search engines and a Q&A programming website to find relevant information, and presented the recommendation directly within an Integrated Development Environment (IDE). Hariri et al. proposed a recommendation system for helping requirement discovery [24]. It recommends relevant topics for requirement discussion and recommends expert stakeholders for each topic discussion.

Different from the above works, we perform different kinds of recommendations. First, we recommend API libraries given a list of already used libraries. Second, we recommend API methods given a feature request and a list of used libraries. Last bust not least, we recommend how to transform code in one version to work in another version.

Chapter 3

Automated Library Recommendation

3.1 Introduction

Third-party libraries are an integral part of many software projects [7, 30, 64]. For example, we have investigated 1008 projects of substantial size from GitHub. We found that 93.3% of them use third-party libraries, at an average of 28 third-party libraries each. The use of such libraries allows the developer to write less code and to focus on the parts of the code that are specific to the project. The use of third-part libraries also reduces the need for testing.

Today, many third-party libraries are readily available to software developers, from repositories such as the Maven repository.[1] Still, effectively using these libraries remains a challenge for developers, because they may not become aware of new libraries as they are released. Developers may thus be led to "re-implement the wheel". An approach is needed to bridge the gap between the many third-party libraries that are available and the developers that need to use them.

In this work, we propose a technique that automatically recommends libraries for a particular project. Given the set of libraries that the project has used, our tech-

[1] `repo1.maven.org`, `http://search.maven.org`

nique recommends other libraries that are potentially useful for it. Our technique follows a hybrid approach that combines association rule mining and collaborative filtering. The association rule mining component extracts libraries that are commonly used together. The component then rates each of the libraries based on their likelihood to appear together with the currently used libraries. The collaborative filtering component works on the assumption that similar projects are likely to share similar third party libraries. The component analyzes the libraries that are used by the n most similar projects. It then rates each of the libraries based on how many of the top-n most similar projects use it. Our technique finally aggregates the recommendations made by the association rule mining and collaborative filtering components.

A number of previous studies have proposed approaches to recommend library methods to be used in a particular context, e.g., [44, 91]. Our work differs from this previous work in terms of the level of granularity considered. While previous approaches recommend a particular method to be used in a particular context, we target the problem of recommending an entire library (e.g., an entire jar file in the case of a Java project). Past approaches assume that the set of relevant libraries is already known to the developer and it is only the methods in these libraries that are unknown. Our work does not make this assumption, and thus complements these existing studies. Indeed, our approach could be deployed first to recommend particular libraries that will interest developers. These results could then be fed to existing approaches to recommend particular methods to be used in different contexts.

To evaluate the effectiveness of our approach, we have downloaded a few hundred Java projects of substantial size ($\geq$ 10,000 lines) from GitHub[2] and investigated the libraries that these projects use. We observe that these projects make use of a substantial number of third-party libraries and thus are appropriate subjects of our study. Evaluating our approach on projects that use many libraries ensures that

[2] `https://github.com/`

it is able to recommend libraries to *real* projects that *need* third-party libraries. We then use ten-fold cross validation to evaluate our approach. For this, we divide the dataset into ten parts. Nine parts are used as training data, and one part is used as test data. The results over the ten iterations are aggregated. To evaluate our results, we use recall rate@5 and recall rate@10, which are often used as evaluation measures [53, 71, 82]. In our experiments we achieve recall rate@5 and recall rate@10 of 0.852 and 0.894, respectively.

The contributions of our work are as follows:

1. We identify a new problem of library recommendation: Given a set of libraries that a project uses, recommend other libraries that are potentially useful for it.

2. We propose a hybrid technique based on association rule mining and collaborative filtering to recommend libraries that a project can use based on the libraries that the project already uses.

3. To test the effectiveness of our approach, we investigate the third-party libraries used by a few hundred projects. Our experiment shows that our approach is effective and can achieve recall rate@5 and recall rate@10 of 0.852 and 0.894, respectively.

The structure of this chapter is as follows. We formally define the problem and provide an illustrative example in Section 3.2. We describe the base algorithms in Section 3.3. We then present our proposed approach in Section 3.4. Next, we describe our experiments in Section 3.5. We conclude in Section 3.6.

3.2 Problem Definition & Illustrative Example

In this section, we define our problem and illustrate it by a motivating example.

Problem Definition. We define the library recommendation problem as follows. Let *allLibraries* be the set of all libraries that are available, *usedLibraries* be the set of libraries that are currently used in the software project, *usefulLibraries* be the set of libraries that are useful for the project, and *rec-Libraries* be the set of libraries that are to be recommended. The goal of our approach is to find the a set *recLibraries* that satisfies the following conditions:

1. $recLibraries \subseteq allLibraries$
2. $recLibraries \bigcap usedLibraries = \emptyset$
3. $recLibraries \bigcap usefulLibraries \neq \emptyset$

The recommended set of libraries needs to be a subset of all available libraries and should not contain any library that is currently used in the project (there is no need to recommend a library that is already used). Finally, we want to recommend libraries that are useful.

Illustrative Example. To illustrate the issues involved, we provide a motivating example. Consider a project named *openpipe* that uses the following libraries: *logback-classic*, *spring-context*, *lucene-core*, *commons-collections*, *commons-dbcp*. Now, consider the database shown in Table 3.1 which maps a set of projects to the libraries that they use. We want to recommend libraries to *openpipe* based on this database.

One approach to recommend libraries is to look for library usage patterns. From the database, one can see that most projects that use *commons-collections* also use *commons-lang* libraries - for projects *easysoa*, *red5-mavenized*, and *thucydides* this is the case. We can thus infer a *library usage pattern*: *commons-collections* $\rightarrow$ *commons-lang*. Based on this usage pattern, since *openpipe* uses *commons-collections* but does not use *commons-lang*, we would recommend *commons-lang* to *openpipe*.

Another approach to recommend libraries is to look for projects that are similar to *openpipe* in Table 3.1 and investigate the set of libraries that they use. We measure the similarity of two projects based on the set of libraries that are used in common between the two projects.[3] Comparing the complete set of libraries already used by *openpipe* with those of the projects in database, we find that there are two projects that use a similar set of libraries: *easysoa* and *fuse*. These projects share the following libraries with *openpipe*: *lucene-core*, *logback-classic*, and *spring-context*. Furthermore, *easysoa* and *fuse* also use a library that is not used in *openpipe*, namely *commons-httpclient*. Thus, we would recommend *commons-httpclient* to *openpipe*.

We propose to automate the above approaches for recommending libraries to a project. We automate the first approach by using association rule mining to extract library usage patterns. We automate the second approach by leveraging a collaborative filtering technique. We then combine these two approaches to recommend libraries.

Table 3.1: Example Project Database

Project	**Libraries**
easysoa	lucene-core, commons-httpclient, logback-classic, spring-context, commons-collections, commons-lang
fuse	commons-httpclient, camel-core, lucene-core, logback-classic, aether-util, spring-context
red5-mavenized	groboutils-core, commons-collections, ehcache, jta, commons-lang, catalina
histone-java	mockito-all, reflections, spring-context, cglib, joda-time, xstream
thucydides	logback-classic, commons-collections, spring-context, commons-lang, opencsv, groovy

3.3 Preliminaries

In this section, we describe several techniques that are used in our approach: *frequent itemset mining* [1], *association rule mining* [1], and *collaborative filtering* [74]. Frequent itemset mining is the first step of association rule mining.

[3]McMillan et al. use API calls as semantic anchors to measure the similarity between projects [47]; we use a similar idea.

3.3.1 Frequent Itemset Mining

Frequent itemset mining takes as input a transaction database (i.e., a multi-set of transactions), where each transaction is a set of items, and outputs sets of items (a.k.a. itemsets) that appear frequently (i.e., each frequent itemset is a subset of many transactions) in the database. In our setting, a transaction is the set of third party libraries that are used by a project. We refer to the number of transactions that contain all of the elements of an itemset I as the *frequency* of I, denoted as $\mathit{freq}(I)$. The *support* of an itemset I is defined as:

$$sup(I) = \frac{\mathit{freq}(I)}{N}$$

where N is the number of transactions in the transaction database. An itemset is *frequent* if its support is no less than *minsup*, where *minsup* is a user-defined minimum support threshold.

Example 1. Consider a project database shown in Table 3.1. Each project can be considered as a transaction. If *minsup* is e.g., 0.5, then $I = \{commons\text{-}collections,$ $commons\text{-}logging\}$ is a frequent itemset in these transactions, computed as follows. I appears in 3 transactions (easysoa, red5-mavenized, thucydides), so $\mathit{freq}(I)$ is 3. Since the number of transactions in the database (N) is 5, $sup(I)$ is 0.6, which is greater than *minsup*.

3.3.2 Association Rule Mining

In addition to frequent itemsets, another kind of pattern can be extracted from the transaction database. For example, from the database shown in Table 3.1, we can infer: "if a project uses $commons\text{-}collections$, then the project is likely to also use $commons\text{-}lang$" because all of the transactions that contain $commons\text{-}collections$ also contain $commons\text{-}lang$. This kind of pattern is referred to as an *association rule*. An association rule is an "if/then" rule that captures a relationship between two itemsets X and Y in the database. It can be written as:

$$X \rightarrow Y$$

where X is the pre-condition of the rule and Y is the post-condition of the rule. The pre-condition is a statement that must be satisfied for the rule to be applied, whereas the post-condition is the result if the pre-condition is met.

We are interested in association rules that apply to many transactions in the database. For this, we use a metric referred to as *support*. The support of an association rule R = $X \rightarrow Y$, denoted as $sup(R)$, is the proportion of transactions in the database that contain $X \bigcup Y$. We also need to measure the likelihood that a rule is true. For this purpose, we use a metric referred to as *confidence*. The confidence of a rule R with pre-condition X and post-condition Y (i.e., $R = X \rightarrow Y$) is defined as follows:

$$conf(R) = \frac{freq_{X \bigcup Y}}{freq_X}$$

Association rule mining extracts all rules that satisfy user-defined minimum support (*minsup*) and confidence (*minconf*) thresholds. We refer to such rules as *significant association rules*. Association rules can be generated from frequent itemsets. We can enumerate all possible pairs of frequent itemsets where one is a subset of another. Consider A and B where A and B are frequent itemsets and A is a subset of B. Then, the generated association rule R is of the form:

$$A \rightarrow B \setminus A$$

The support and confidence of the rule R can be computed from the supports of its constituent itemsets as follows:

$$sup(R) = sup(B)$$

$$conf(R) = \frac{sup(B)}{sup(A)}$$

Example 2. We refer to Table 3.1. Given $minsup$ = 0.5, frequent itemsets $A = \{commons\text{-}collections\}$ and $B = \{commons\text{-}collections, commons\text{-}lang\}$. Frequent itemsets A and B form the rule $R = commons\text{-}collections \rightarrow commons\text{-}lang$. The support of R is the same as $sup(B)$, which is 0.6. Since $sup(A)$ is 0.6, the confidence of the rule is 1.0.

3.3.3 Collaborative Filtering

Collaborative filtering is an automatic technique to make predictions about an entity based on information collected about other similar entities. Collaborative filtering has been used in many real systems, including environmental sensing, financial services, electronic commerce, web applications, etc. [69]. One popular implementation of collaborative filtering in the context of web applications is the recommendation system developed by Amazon.com for recommending new items to the users in their website [35].

A basic method to perform collaborative filtering is by finding the nearest neighbors of the target entity. A target entity is compared with all other entities and a list of most similar entities based on a distance metric is produced. The similarities among the entities are used as a basis for making predictions about the entity. In our setting, an entity is a project, and the prediction task is the prediction of libraries that are useful for the project.

3.4 Proposed Approach

In this section, we first provide a birds-eye-view description of our overall framework.We then zoom in to the various components of our framework. We describe the association rule mining component.We then present the collaborative filtering component. The aggregator component which combines the result of the association rule mining and collaborative filtering components.

3.4.1 Overall Framework

Figure 4.2 shows the overall framework of our recommendation system referred to as $LibRec$. It has two major components: $LibRec^{RULE}$ and $LibRec^{COLLAB}$. $LibRec^{RULE}$ recommends libraries by mining library usage patterns expressed as association rules. $LibRec^{COLLAB}$ recommends libraries by investigating the set of libraries that are used by similar projects, using a nearest-neighbor-based collaborative filtering approach. Our framework consists of two phases: *training* and *deployment*.

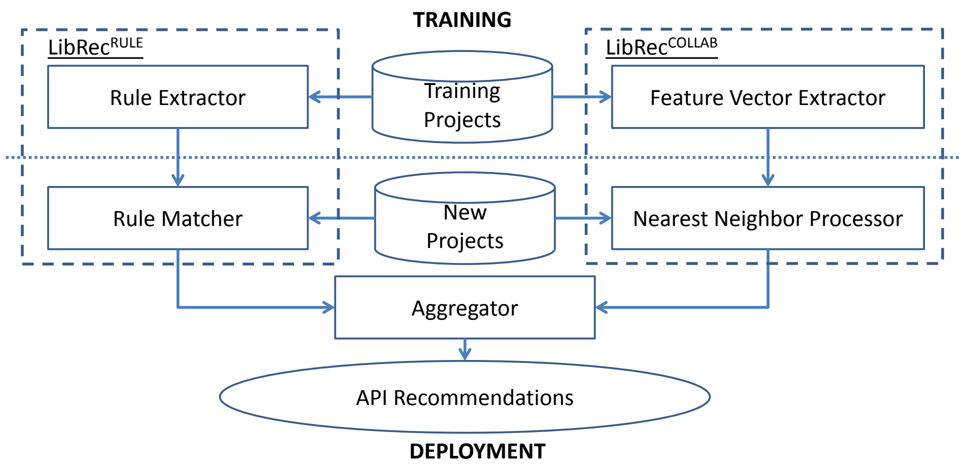


Figure 3.1: Our Recommendation Framework $LibRec$

In the training phase, our system infers models needed for the deployment phase. These models are extracted from a training dataset ($TrainingProjects$). $TrainingProjects$ is a set of projects, along with the names of the third-party libraries used by each of them. Models are extracted by the following sub-components:

1. $RuleExtractor$, a sub-component of $LibRec^{RULE}$, extracts association rules that capture library usage patterns from *Training Projects*.

2. $FeatureVectorExtractor$, a sub-component of $LibRec^{COLLAB}$, extracts a vector of feature values from the set of libraries that each project uses. Each

feature corresponds to a library and its value is 1 if the library is used by the project and 0 otherwise.

The association rules and vectors are models that are provided to the deployment phase.

In the deployment phase, our system recommends libraries to new projects ($NewProjects$) based on the models extracted from the training phase. $NewProjects$ is a set of new projects along with the names of third-party libraries already used by each of them. The following sub-components use the models to recommend libraries:

1. $RuleMatcher$, a sub-component of $LibRec^{RULE}$, takes each new project $p \in NewProjects$ and the association rules extracted in the training phase as inputs. It matches the libraries used in the project with the rules. It makes recommendations based on the post-conditions of the matching rules.

2. $NearestNeighborProcessor$, a sub-component of $LibRec^{COLLAB}$, takes each new project $p \in NewProjects$ and the feature vectors extracted in the training phase as inputs. It then constructs a feature vector for p and calculates the distance between this feature vector and each of the feature vectors extracted from the training phase. The most similar vectors and their corresponding projects (i.e., the nearest neighbors) are identified. Recommendations are made based on the libraries used by the nearest neighbors.

Both $RuleMatcher$ and $NearestNeighborProcessor$ output a list of recommended libraries along with their recommendation scores. The $Aggregator$ component combines these two lists into a new list with the final recommendation scores. The libraries with the highest scores will be recommended.

We now describe each of the components in more detail.

3.4.2 $\mathit{LibRec}^{\mathit{RULE}}$ Component

Our $\mathit{LibRec}^{\mathit{RULE}}$ component recommends libraries based on library usage patterns. It consists of the $\mathit{RuleExtractor}$ sub-component in the training phase and the Rule-$\mathit{Matcher}$ sub-component in the deployment phase.

3.4.2.1 $\mathit{RuleExtractor}$

This sub-component employs association rule mining to mine library usage rules. In $\mathit{TrainingProjects}$, each project uses a set of third-party libraries. We treat each project as a transaction where the items in the transactions are the third-party libraries that it uses.

Traditional association rule mining (see Section 3.3) extracts *all* rules that satisfy the minimum support and confidence thresholds from the set of *all* frequent itemsets. However, often, too many rules are extracted. Association rule mining can thus become very slow. To address this issue, we observe that not all of the rules are necessary; some of the rules can be combined to construct a compact set of association rules.

Example 3. Consider the following rules: $\mathit{R1} = \mathit{log4j} \rightarrow \mathit{commons}\text{-}\mathit{logging}$, $\mathit{R2} = \mathit{log4j} \rightarrow \mathit{slf4j}\text{-}\mathit{api}$, and $\mathit{R3} = \mathit{log4j} \rightarrow \mathit{commons}\text{-}\mathit{logging}, \mathit{slf4j}\text{-}\mathit{api}$. All of these rules have support 0.6 and confidence 1.0. Rules $\mathit{R1}$ and $\mathit{R2}$, however, are covered by rule $\mathit{R3}$ and they have the same support and confidence, and thus they are not actually needed.

To construct a compact set of association rules, we use two special subsets of *all* frequent itemsets, referred to in the literature as *closed itemsets* and *generators* [55]. A closed itemset is a frequent itemset with no superset having the same support as itself. A generator is a frequent itemset with no subset having the same support as itself. Algorithms have been developed to mine these closed itemsets and generators directly [55, 33], without the need to mine all frequent itemsets.

Example 4. Consider the following two frequent itemsets: $I1 = \{\mathit{log4j}\}$ and $I2 =$

$\{log4j, commons\text{-}logging, slf4j\text{-}api\}$. $I1$ has support of 3 and does not have any subset having the same support. Thus, $I1$ is a generator. $I2$ has support of 3 and does not have a superset having the same support. Thus, $I2$ is a closed itemset. Note that $I1$ is not a closed itemset and $I2$ is not a generator as $I1 \subset I2$ and $sup(I1) = sup(I2)$.

We construct the compact set of association rules from these two subsets following Bastide et al. [6]. We define a compact set of association rules in Definition 1.

Definition 1 (Compact Assoc. Rules (ARules^{Compact})) *Consider a set of generators GEN, a set of closed itemsets $CLOSED$, a minimum support $minsup$ and a minimum confidence $minconf$. The compact set of association rules is the following set:*

$$\begin{aligned}\{r = pre \rightarrow post | (pre \in GEN) \wedge \\ (pre \bigcup post \in CLOSED) \wedge \\ sup(r) \geq minsup \wedge conf(r) \geq minconf\}\end{aligned}$$

The set of compact rules is the set of frequent rules whose pre-condition is a generator and where the difference of the pre- and post-conditions is a closed itemset.

Example 5. Referring to Example 4, we can construct a compact rule from $I1$ and $I2$. The pre-condition of the compact rule is $I1 = \{log4j\}$ and the post-condition of the compact rule is $I2 \setminus I1 = \{commons - logging, slf4j - api\}$. Thus, we can the construct compact rule $C = log4j \rightarrow commons - logging, slf4j$ $-api$.

The set of association rules obtained from closed itemsets and generators is potentially much smaller than the complete set of rules. The algorithm presented in Figure 3.2 constructs this compact set of association rules. The algorithm first mines the set of closed itemsets and generators (lines 7-8). We use the algorithm

Zart [73] to mine these closed patterns and generators.[4] Next, the algorithm iterates over all of the closed itemsets and generators (lines 9-10). If the generator (*Gen*) is the subset of the closed itemset (*Item*), a rule is constructed (lines 11-22). *Gen* is the pre-condition of the rule and $Item \setminus Gen$ is its post-condition (lines 14-15). We then compute the support and confidence of the rule (lines 16-17). The constructed rule is added to the compact set of association rules if it satisfies the $minconf$ threshold (lines 18-19). There is no need to check the $minsup$ threshold as the closed itemsets are frequent. At the end, all the generated rules are stored for use in the deployment phase.

```
 1: Input:
 2:    $TrainingProjects$ = set of projects with third party
                          libraries used by each of them
 3:    $minconf$ = minimum confidence threshold
 4: Output:
 5:    Compact set of association rules
 6: Method:
 7: Let $ClosedItems$ = set of closed itemsets mined from
                     $TrainingProjects$
 8: Let $Generators$ = set of generators mined from
                    $TrainingProjects$
 9: Let $CompactRules = \{\}$
10: for all $Item \in ClosedItems$ do
11:    for all $Gen \in Generators$ do
12:       if $Gen \subset Item$ then
13:          Let $Rule = \{\}$
14:          $Rule.PreCondition = Gen$
15:          $Rule.PostCondition = Item \setminus Gen$
16:          $Rule.Support = Item.Support$
17:          $Rule.Conf = Item.Support / Gen.Support$
18:          if $Rule.Conf \geq minconf$ then
19:             add $Rule$ to $CompactRules$
20:          end if
21:       end if
22:    end for
23: end for
24: return $CompactRules$
```

Figure 3.2: Mining a Compact Set of Association Rules

[4]http://www.philippe-fournier-viger.com/spmf/index.php.

3.4.2.2 $RuleMatcher$

In the deployment phase, for each new project p to receive library recommendations, $RuleMatcher$ gets the list $currentLib$ of the libraries that it currently uses, and then matches this list against the association rules $libRules$ generated by the $RuleExtractor$ component. A rule *matches* $currentLib$ if its precondition is a subset of $currentLib$. $RuleMatcher$ then recommends libraries, based on the post-conditions of the matching rules.

We assign a score to assess the suitability of a library to a new project p. Informally, the *rule-based recommendation score* for a library A is the highest confidence of any matching rule whose post-condition contains A. This score is computed by the following formula:

$$\begin{aligned} RecScore^{RULE}(A) = MAX(0, MAX_{R \in RMatched(A)}.conf(R)) \\ RMatched(A) = \{(R = X \rightarrow Y) \in libRules \mid \\ X \subseteq currentLib \wedge A \in Y\} \end{aligned}$$

In the above equation, $RMatched(A)$ is the set of rules whose pre-condition is a superset of $currentLib$ and whose post-condition contains A. If the set $RMatched(A)$ is empty, the recommendation score of A is 0. $RecScore^{RULE}$ ranges from 0 to 1. The libraries with the highest recommendation scores are the most suitable libraries based on the mined association rules.

Example 6. Suppose a project has a set of libraries $P = \{a, b, c\}$ and we have the following rules: $R1 = e \rightarrow d$ with $conf(R1) = 1.0$, $R2 = a, b \rightarrow f, g$ with $conf(R2) = 0.9$, and $R3 = a \rightarrow f$ with $conf(R3) = 0.8$. $R2$ and $R3$ match P because their pre-conditions are a subset of P. We then want to compute $RecScore^{RULE}$ for the library f. Both $R2$ and $R3$ contain f in their post-conditions. Thus, $R2$ and $R3$ are matching rules. $RecScore^{RULE}(f)$ will then be the maximum of 0, 0.9 ($conf(R2)$), and 0.8 ($conf(R3)$), which is 0.9.

The pseudocode for the matching process is shown in Figure 3.3. At lines 8-9, for each association rule in $libRules$, we check whether $curLibraries$ is a superset of the pre-condition of the rule. If it is, we iterate over the items in the rule's post-condition and add them to the list of recommended libraries ($recLibraries$) (lines 10-18). We iteratively update the recommendation scores of the libraries in $recLibraries$. Whenever a matching rule of higher confidence containing a recommended library is encountered, we update the recommendation score of the library (lines 14-16).

```
 1: **Input:**
 2:    $curLibraries$ = libraries that the target project uses
 3:    $libRules$ = association rules
 4: **Output:**
 5:    Recommended libraries w. recommendation scores
 6: **Method:**
 7: **Let** $recLibraries$ = { }
 8: **for all** $Rule \in libRules$ **do**
 9:    **if** $curLibraries \subseteq Rule.PreCondition$ **then**
10:       **for all** $A \in Rule.PostCondition$ **do**
11:          **if** $A \notin curLibraries$ **then**
12:             **add** $A$ **and** $A.Conf$ **to** $recLibraries$
13:          **else**
14:             **if** $recLibraries[A] < A.Conf$ **then**
15:                $recLibraries[A] = A.Conf$
16:             **end if**
17:          **end if**
18:       **end for**
19:    **end if**
20: **end for**
21: **return** $recLibraries$
```

Figure 3.3: Rule Matching Procedure

3.4.3 $LibRec^{COLLAB}$ Component

Our $LibRec^{COLLAB}$ component recommends libraries based on those that are used by similar projects, following a nearest-neighbor-based collaborative filtering approach. We measure the similarity of two projects based on their set of commonly used libraries. $LibRec^{COLLAB}$ consists of the $FeatureVectorExtractor$ sub-component in the training phase and the $NearestNeighborProcessor$ sub-component in the deployment phase.

3.4.3.1 $FeatureVectorExtractor$

This component converts the list of libraries used by each project in $Training$ $Projects$ to a feature vector. Let aL be the set of all libraries arranged in alphabetical order of their names. Each library can then be assigned a unique index in aL and referred to as $aL[i]$. The feature vector of project A, denoted as $V(A)$, is defined as follows:

$$V(A) = (ind(aL[0], A), \ldots, ind(aL[|aL|], A))$$

$$ind(L, A) = 1,\ If\ \ A\ uses\ library\ L$$

$$0,\ Otherwise$$

3.4.3.2 $NearestNeighborProcessor$

Given a new project to receive library recommendations, $NearestNeighborPr$-$ocessor$ converts the list of libraries that the project uses into a feature vector in the same manner as was done by the $FeatureVectorExtractor$ component. It then calculates the distance between this feature vector and the feature vectors of projects in $TrainingProjects$. We use cosine similarity as the metric to compute this distance [45]. The cosine similarity of a new project New and an existing project $Existing$ in $TrainingProjects$ is:

$$Cosine(New, Existing) = \frac{V(New) \cdot V(Existing)}{|V(New)||V(Existing)|}$$

In the above equation, $\cdot$ denotes dot product, and $|V(i)|$ denotes the size of a vector, which is defined as the square root of the sum of the squares of its constituent elements.

We rank the projects in $TrainingProjects$ based on their cosine similarity scores. The higher the cosine similarity score is, the more similar a training project is to the new project. Therefore, we pick the top-n projects with the highest cosine similarity scores as the nearest neighbors for the new project. In the implementation, we sort the projects based on their cosine similarity scores followed by their names. If there are projects with rank greater than n that have the same cosine similarity score as the n-th project, we group the projects having this score, and randomly select projects from this group, to result in the final n nearest neighbors.

Our next step is to compute a recommendation score for each library. We collect all of the libraries that are used by projects in the n nearest neighbors and compute the score for each library. Given a library A, we compute the *collaborative-filtering-based recommendation score* for A as follows:

$$RecScore^{COLLAB}(A) = \frac{NNCount_{Lib}(A)}{n}$$

In the equation above, $NNCount_{Lib}(A)$ is the number of nearest neighbor projects that use library A and n is the number of nearest neighbors. $RecScore^{COLLAB}$ scores range from 0 to 1. The library with the highest score of $RecScore^{COLLAB}$ is considered to be the most the most suitable library.

Example 7. Consider a project that has 3 nearest neighbors as follows: $P1 = \{ant, jsp\text{-}api, junit\}$, $P2 = \{hsqldb, junit, commons\text{-}lang\}$, and $P3 = \{log4j, jetty, gson\}$. $junit$ is used in 2 out of 3 nearest neighbor projects. So, $NNCount_{Lib}$-$(junit)$ is 2 and $RecScore^{COLLAB}(junit)$ is 0.67.

3.4.4 Aggregator Component

The *Aggregator* component combines $RecScore^{RULE}$ and $RecScore^{COLLAB}$ to get an overall recommendation score, denoted as $RecScore$. The overall recommendation score of a library A is defined as follows:

$$RecScore(A) \quad = \quad \alpha \ \times \ RecScore^{RULE}(A) \ + \quad \beta \ \times \ RecScore^{COLLAB}(A)$$

In the equation above, α and β represent weights for the two recommendation strategies. When $\alpha + \beta = 1$, $RecScore$ ranges from 0 to 1. By default, we set α and β to 0.5. The top-k libraries with the highest $RecScore$s are recommended to developers – again ties are randomly broken.

Example 8. Suppose that we have a library called z. Let $RecScore^{RULE}(z)$ = 0.8 and a $RecScore^{COLLAB}(z)$ = 1.0. $RecScore(z)$ is 0.9.

3.5 Experiments & Analysis

In this section, we describe our dataset, followed by our evaluation measures and procedure. We then present our research questions and the results of our experiments. Finally, we discuss some threats to validity.

3.5.1 Dataset

To construct the dataset, we first randomly collected a few thousand Java projects from GitHub.[5] We then filtered the collected projects based on the following criteria:

1. **The project contains more than 10,000 lines of code.** This is intended to filter out "toy projects". We counted the line of codes by using SLOCCount,[6] which excludes comments and whitespace.

[5] `http://github.com`
[6] `http://www.dwheeler.com/sloccount`

2. **The project is not a fork of another project in Github.** A fork is essentially a clone of another project at a specific point of time. We do not want to consider both the original and the forked project, or multiple projects that are forked from the same source. The original and forked projects are likely to share the same libraries.

3. **The project is a Maven project.** Maven is a build automation tool. One of its features is the ability to declare library dependencies for a project. Libraries have a unique identifier that can help us to correctly identify the use of the library across multiple projects. We identify Maven projects by checking for the existence of pom.xml files in the project repository. From these xml files, we extracted the *GroupId* and *ArtifactId* of the libraries on which the project depends. The combination of these two identifiers forms a unique identifier for a library stored in the Maven repository.[7]

4. **The project uses at least ten libraries.** We focus on projects that rely on third-party libraries. Our recommendation system has more value for library-intensive projects.

After filtering projects that contain fewer than 10,000 lines of code, are forked from other projects, and are not Maven projects, we are left with 1008 projects. The distribution of the number of libraries used in these projects is shown in Figure 3.4. The minimum, maximum, and average number of libraries used in these projects are 0, 627, and 28.1 projects, respectively. To focus on projects that rely heavily on third-party libraries, we randomly selected 500 projects that use at least ten libraries as our experimental dataset. Projects included in this dataset include popular projects such as Tapestry5 (146.4 kLOC), Sonar (132.3 kLOC), JBoss (499.0 kLOC).[8]

[7]`repo1.maven.org`, `http://search.maven.org`

[8]The list of selected projects is available at: `https://sites.google.com/site/autolibrec/projects`

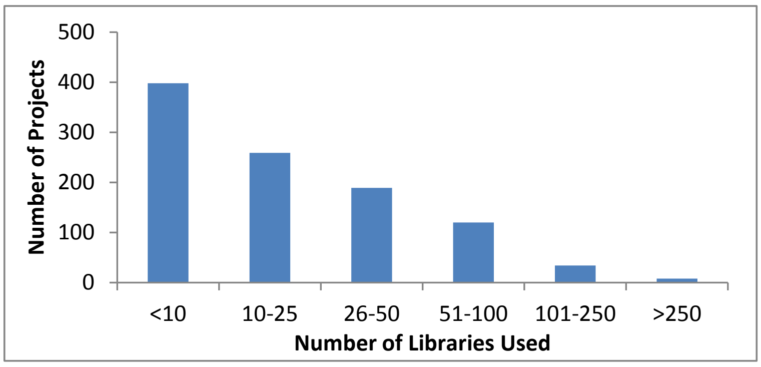


Figure 3.4: Distribution of Library Usage in 1008 Projects

3.5.2 Evaluation Measures and Procedure

We evaluate our experiments using a well known evaluation metric, recall rate@k [53, 54, 66, 71, 82].[9] Consider m target projects that should receive library recommendations. For each project p_i, let the ground truth be the set of libraries GT_i. The recall rate@k of a library recommendation system that recommends a set of top-k libraries R_i for each of the projects p_i, is the proportion of recommendation R_i, in the set of all recommendations R (for all projects), that includes at least one library in the ground truth (i.e., $R_i \bigcap GT_i \neq \emptyset$). We use a small value for k as developers are unlikely to look through a long recommendation list.

We perform ten-fold cross validation to measure the accuracy of our approach. We randomly distribute the dataset into ten equal-sized parts. Each fold consist of nine parts of the dataset as the training data and the remaining part as the testing data. For each project in the test data, we drop half of their libraries and use these as the ground truth. The remaining half are used as inputs to our recommendation approach. This methodology mimics the scenario where a developer knows some of the needed libraries but needs help to find other relevant libraries.

LibRec takes a number parameters: $minsup$, $minconf$, n (i.e., number of neighbors), α (i.e., weight of the $LibRec^{RULE}$), and β (i.e., weight of the $LibRec^{COLLAB}$). We set $minsup = 0.1$ and $minconf = 0.8$. We chose $minsup = 0.1$ because we

[9]Note that precision rate@k is not defined and is not used in past studies [53, 54, 66, 71, 82].

do not want to miss specific rules that only exist in a small portion of the dataset. We chose $minconf = 0.8$ because we want the rule to have a high likelihood of being followed in a software project. We set the parameter n to 20, and the other parameters to their default values i.e., $\alpha = \beta = 0.5$.

3.5.3 Research Questions

We consider the following three research questions:

RQ1 How accurate is our proposed approach in recommending libraries to client applications?

RQ2 What are the effects of the various components and parameters of our approach on the overall accuracy?

RQ3 What is the impact of the various experimental settings on the overall accuracy?

3.5.4 RQ1: Accuracy of the Proposed Approach

We have performed a ten-fold cross validation on the 500 projects. The experiment shows that *LibRec* achieves a recall rate@5 of 0.852 out of 1.

3.5.5 RQ2: Effectiveness of Various Components and Parameter Settings

To answer this research question, we investigate the effectiveness of the two major components of *LibRec* and the sensitivity of *LibRec* to the various parameter settings.

Effectiveness of the Individual Components. We investigate the how well our individual components $LibRec^{RULE}$ and $LibRec^{COLLAB}$ work separately. The result is shown in Table 3.2. Our $LibRec^{COLLAB}$ component performs better than our $LibRec^{RULE}$ component. Still, both of their recall rates are lower than that of *LibRec*. Thus, combining the two components is beneficial, as it improves accuracy.

Table 3.2: Effectiveness of Individual Components

Component	**Recall Rate@5**
$LibRec^{RULE}$	0.702
$LibRec^{COLLAB}$	0.800

Effect of the Varying Number of Neighbors. We next investigate the sensitivity of our approach to the number of nearest neighbors taken into account. In practice, developers might not know the best number, and thus it is best if our approach is robust on different numbers of nearest neighbors, within a particular range. We vary the number of nearest neighbors from 5 to 25 and show the accuracy of *LibRec* in Table 3.3. We find that the accuracy of our approach is relatively stable across different numbers of nearest neighbors (differences are less than 0.012). This shows the robustness of our approach.

Table 3.3: Effect of Varying the Number of Nearest Neighbors

Number of Nearest Neighbors	**Recall Rate@5**
5	0.840
10	0.848
15	0.850
20	0.852
25	0.848

Effect of Varying *minsup* and *minconf*. We next investigate the effect of varying *minsup*. As shown in Table 3.4, there is a small drop in accuracy when *minsup* is increased from 0.1 to 0.3 (about 0.05 reduction in recall rate@5). Increasing *minsup* eliminates some high confidence rules that apply to only a few projects. This reduces the effectiveness of our approach on these projects. The recall rate is relatively stable when we increase *minsup* from 0.3 to 0.4 and 0.5.

We also investigate the effect of varying *minconf*. We notice that on very high *minconf* settings, there is a small drop in accuracy (about 0.07 drop in recall rate@5, when *minconf* is increased from 0.9 to 1.0). Raising the required confidence too high can cause good rules that might not apply in a few cases to be omitted. Note that *LibRec* uses the confidence of a matching rule to compute the rule-based recommen-

Table 3.4: Effect of Varying *minsup*

minsup	**Recall Rate@5**
0.1	0.852
0.2	0.816
0.3	0.796
0.4	0.800
0.5	0.800

dation score. Thus, we notice that reducing *minconf* from 0.9 to 0.8 results in little change in accuracy as *LibRec* takes a matching rule with the *highest* confidence.

Table 3.5: Effect of Varying *minconf*

minconf	**Recall Rate@5**
0.80	0.852
0.85	0.852
0.90	0.848
0.95	0.824
1.00	0.778

3.5.6 RQ3: Effectiveness of Various Experimental Settings

To answer this research question, we investigate the sensitivity of *LibRec* to two experimental settings.

Effect of Varying Training Set Size. We vary the training set size by varying the value of k in k-fold cross validation. As k increases, the training set size also increases. The result for this experiment is shown in Table 3.6. We notice that the average recall rate@5 does not vary much on the training set size (differences are at most 0.014).

Table 3.6: Effect of Varying Training Set Size

k **Fold**	**Recall Rate@5**
2	0.840
4	0.848
6	0.840
8	0.854
10	0.852

Effect of Changing Recall Rate@k. Finally, we investigate the effect of changing the value of k for recall rate@k. Intuitively, a larger k results in a higher recall rate. As shown in Table 3.7, the recall rate@1 is 0.616, and recall rate@3 is 0.804 which means, for most cases, correct recommendations appear early in the recommendation list.

Table 3.7: Effect of Varying Recall Rate@k

Recommendation Size (i.e., k)	**Recall Rate @ k**
10	0.894
7	0.866
5	0.852
3	0.804
1	0.616

3.5.7 Threats to Validity

Threats to internal validity refers to experimenter bias. Most of our experimental process is automated and randomized. Thus we believe there is little experimenter bias.

Threats to external validity refers to the generalizability of our findings. Our dataset consists only of open source Java projects. Moreover, we pick only Java projects that use Maven. In practice, only a subset of Java developers use Maven to develop their applications. Even so, Maven is a popular tool in the Java developer community. We expect big projects that use many libraries to use Maven or similar tools to help manage their build process. We have already tried to minimize this threat by investigating 500 random projects. In the future, we plan to reduce this threat further by adding more projects.

Threats to construct validity refers to the suitability of our evaluation measure. Currently, we use recall rate@k to measure the effectiveness of our approach. This is a well known measure that is used in many past studies, e.g., [53, 71, 82].

3.6 Conclusion

Third party libraries can help to reduce software system development time. Developers can reuse the libraries to code some parts of the system rather than implementing them by themselves. Using well tested libraries also makes the system more reliable. Many third party libraries are publicly available. However, the large number of libraries makes it hard for developers to pick the relevant libraries that can improve their productivity.

We propose an automated technique to recommend relevant libraries to developers. Our approach combines association rule mining techniques and collaborative filtering to perform the recommendation. Based on the libraries used by other projects, we recommend a number of likely relevant libraries to developers of a target project. We have evaluated our approach on 500 open source Java projects hosted on GitHub. Our approach achieves a promising results with recall rate@5 and recall rate@10 of 0.852 and 0.894 respectively.

In the future, we plan to include more projects to further validate our results. We also plan to develop a better approach that can increase the recall rate. To achieve this, we plan to analyze cases where our approach and individual components are ineffective and make appropriate modifications to our approach. One approach could be to consider not only libraries but also abstractions of libraries, e.g., the domain that they address. Another approach could be to consider nonfunctional properties of the projects, such as the time at which they were developed. Yet another approach would be to take into account the textual descriptions of libraries, by employing advanced NLP techniques, e.g., [10, 81], or to analyze the usage specifications of these libraries inferred by specification mining techniques, e.g., [15, 29, 39, 40, 41, 68]. We would also like to extend our approach to be able to recommend libraries to projects that only use a small number of libraries or do not use any libraries at all.

In terms of our experimental method, we plan to experiment with various experimental settings, e.g., considering different numbers of projects being dropped, considering different number of libraries used, etc. We also plan to integrate our proposed approach in an IDE (e.g., Eclipse) and perform a user study.

Finally, we also want to integrate our approach with techniques that recommend specific methods to use in a library, e.g., [44, 61, 75, 76] and to recommend specific library versions.

Chapter 4

Improving Library Recommendation using Textual Content and Matrix Factorization

4.1 Introduction

Leveraging third party libraries is a common practice when developing a software project. The main purpose is to prevent developers from needing to reinventing the wheel and thus allowing them to focus their attention on the main features of the developed software project. From an economic perspective, reusing libraries often reduces overall development cost. By freeing developers from implementing functionalities provided by the libraries, a software project can be finished faster, thereby reducing development time cost [34, 50]. Also, by opting to make use of third party libraries rather than implementing similar functionalities themselves, bug fixing costs would be reduced since third party libraries are typically well tested.

For common programming languages (i.e., Java), the number of third party libraries is growing and their usage in many software projects has proliferated [7, 30, 64]. Discovering the libraries has been made easier by the existence of a centralized repository for third party libraries. For libraries written in Java, one such repository is Maven Central Repository[1]. To date, this repository has indexed about 182,019 unique libraries. Certainly, it is impossible for any developers to understand each and every one of them. And even if it were possible, developer time would still be wasted as most of the libraries are guaranteed to be useless for them. Thus, automation is a necessary to effectively recommend libraries to developers.

Our work is an extension of our previous work on automated library recommendation [77], which is presented in Chapter 3. Our previous approach recommends libraries to developers by using collaborative filtering and association rule mining. We build upon our previous work by leveraging the textual content of libraries and combining our previous techniques with a matrix factorization based technique. We follow the same approach as our previous work when applying association rule mining. For collaborative filtering, we employ a more fine grained representation of a software project by leveraging textual content of its adopted third party libraries. We then introduce an additional way for solving the library recommendation problem by using a matrix factorization technique. This technique learns a vector representation for both the software projects and the libraries based on historical library usage from known open-source software projects. These representations can be used to infer how likely a library would be useful for a target software project. Each of the above techniques generates a likelihood score for each library, representing how likely it is that the library would be useful for developing a target software project. We combine the scores from each of the above techniques into a final score and use it to rank third party libraries. We name our proposed approach *LibXplore*.

For evaluating the effectiveness of a library recommendation approach, in our previous work described in Section 3, we simulate initial and historically added libraries by randomly dividing the set of libraries used in the latest version of a software project. One group is considered as the set of initial libraries and the other is considered as the set of historically added libraries. In this work, we mine software repositories to extract a set of libraries that is used in an initial version of

[1]`https://search.maven.org/`

a software project and another set that is used in the latest version. The difference between these two sets is the set of historically added libraries, while those in the first set are the initial libraries. To evaluate our approach, we use these actual sets of initial and historically added libraries. In this way, our evaluation resembles a more realistic scenario and its results signals how well our approach would fare if it was used to recommend libraries to developers.

In our experiment, we use the same dataset used to evaluate our previous work [77]. We further filter the dataset to include only libraries that are still available. Our evaluation is then done through 10-fold cross validation, in which the dataset is divided into roughly ten equal parts. Ten iterations are then performed. For each of them, one part is used as testing data and the other parts are used as training data. Evaluation results over these iterations are then averaged. We report this averaged evaluation metric. We compare our approach with our previous state of the art approach. Our experiment shows the superiority of our approach, achieving Hit@5 of 0.607 and Hit@10 of 0.702, improving over the state of the art by 20.44% and 9.69%, respectively.

The main contributions of our work can be summarized as follows.

1. We propose a new library recommendation approach called *LibXplore*. This approach combines the power of collaborative filtering, association rule mining, and matrix factorization. It considers a set of existing libraries as input, the historical library usage and the textual content from initial libraries as knowledge source.

2. We introduce a more realistic scenario for evaluating library recommendation approach by mining actual set of initial and historically added libraries from version control repository.

3. We empirically evaluate the effectiveness of our approach in a 10-fold cross validation setting. The obtained result shows that our approach beats our previous approach. Our approach achieves Hit@5 of 0.607 and Hit@10 of

0.702, improving over theirs by 20.44% and 9.69%, respectively.

The rest of this chapter is structured as follows. We describe some preliminary materials in Section 4.2. Next, we describe our proposed approach in Section 4.3. We then explain our experiments in Section 4.4. Finally, we conclude and mention future work in Section 4.5.

4.2 Preliminaries

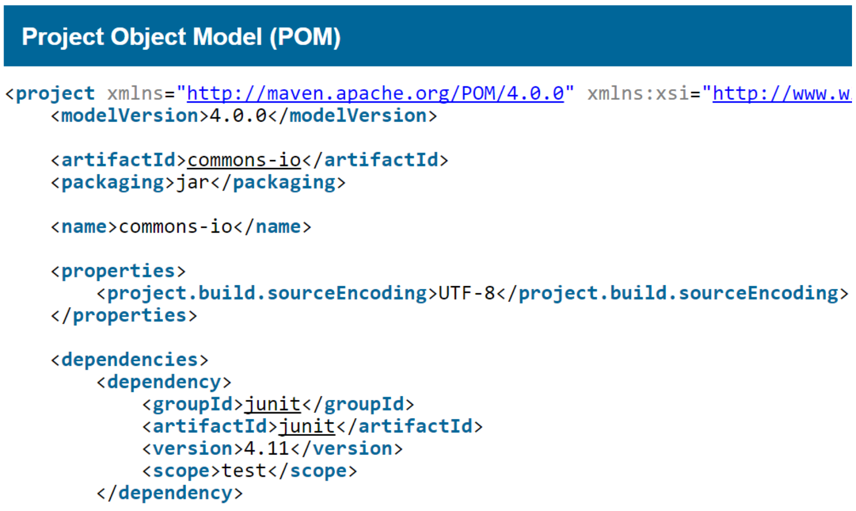
In this section, we describe some background. We first present Maven. Next , we present vector space model.

4.2.1 Maven

Maven is a popular software project management tool for Java programming language. It uses a model called Project Object Model (POM) to manage a project. Figure 4.1 shows an example of a POM file for *commons-io*. As shown, POM can provide a lot of information about a project. For the purpose of this work, our attention is focused only on the dependencies tag inside a POM file. This tag lists required dependencies for a project. It contains information such as *groupId* (typically a name of company/organization), *artifactId* (unique name for an artifact in a *groupId*), *version* (version of the artifact), and *scope* (related to POM lifecycle, indicates in which part of the cycle the dependency is considered). The pair of the *groupId* and the *artifactId* provides a unique identifier for a dependency. This identifier is important for extracting library usage across various software projects.

4.2.2 Vector Space Model

In the vector space model, a document is converted to a bag of words. Each word in the bag becomes an element in a vector. Each element contains a value signifying the importance of the corresponding word in the document. To measure this word



Project Object Model (POM)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w
    <modelVersion>4.0.0</modelVersion>

    <artifactId>commons-io</artifactId>
    <packaging>jar</packaging>

    <name>commons-io</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
```

Figure 4.1: Project Object Model (POM) for *commons-io*

importance, a combination of term frequency and inverse document frequency are commonly used.

Term frequency (TF) is the number of times a word appears in a document. If a word appears often in a document, it is considered to be important for that document. Inverse document frequency (IDF) is the reciprocal of the number of documents in the collection that contain a word. A higher IDF indicates the higher importance of the word since the word is relatively more unique to the corresponding document compared to others. The TF-IDF for a word i in a document D and document collection C is defined as follows.

$$w_{i,D,C} = TF_{i,D} \times IDF_{i,C}$$

$$IDF_{i,C} = \frac{1}{DF_{i,C}}$$

$TF_{i,D}$ is the number of times a word i occurs in document D while $DF_{i,C}$ is the number of documents in C that contains word i.

Given a bag of words of document D in a document collection C, we can generate its vector space model representation by first calculating TF-IDF for each word in the collection using the above formula. We then construct a vector whose size is the number of unique words in C. We assign a TF-IDF weight for the corresponding word dimension in this representative vector. For words that do not occur in D, we assign their TF-IDF weight to zero.

Using the vector space model, we can calculate the similarity between two documents by calculating the similarity between their vector representations. Typically, the similarity is defined in the form of cosine similarity. Given two document vectors V_1 and V_2 with size N, the cosine similarity between them is calculated as follows.

$$Cosine(V_1, V_2) = \frac{\sum_{i=1}^{N} w_{i,V_1} \times w_{i,V_2}}{\sqrt{\sum_{i=1}^{N} w_{i,V_1}^2}\sqrt{\sum_{i=1}^{N} w_{i,V_2}^2}}$$

In the equation, $w_{i,V}$ is the TF-IDF weight for the i^{th} word in vector V.

4.3 Proposed Approach

In this section, we provide an overview of our approach. We then describe components of our approach. We first describe the association rule mining component, the collaborative filtering component, and the matrix factorization component. Results from the above components are then fed to the combiner component.

4.3.1 Overview

We show the overview of our library recommendation approach $LibXplore$ in Figure 4.2. Our approach consists of several components: $LibXplore^{RULE}$, $LibXplore^{CF}$, $LibXplore^{MF}$, and $Combiner$.

$LibXplore^{RULE}$ generates library usage association rules by analyzing *Training Projects*, which contains library usages for known projects. Given a *Target Project*, this component selects association rules that contains the project libraries. It then collects libraries in the post-conditions of the rules and assigns scores to them.

$LibXplore^{CF}$ retrieves *Training Projects*. Each project is represented as a collection of libraries. It then generates a vector representation for each project according to textual contents of the libraries. Given a *Target Project*, $LibXplore^{CF}$ generates the same vector representation. Using this vector, $LibXplore^{CF}$ finds *Training Projects* that are the most similar to the input project. Libraries in the most similar projects are then assigned likelihood scores.

$LibXplore^{MF}$ retrieves *Training Projects* and a *Target Project*. It then creates project-library usage matrix that indicates whether a project uses a particular library. Based on this matrix, $LibXplore^{MF}$ infers latent features for projects and libraries. These features are learned so that interaction between the two can predict the likelihood score of a library to be used in a given project.

Given the likelihood scores from the three components, $Combiner$ combines them into a single likelihood score. This score is used to rank libraries to be recommended. The ranked libraries are then output as *New Library Recommendations*.

Details about each of the above components are described in the following sections.

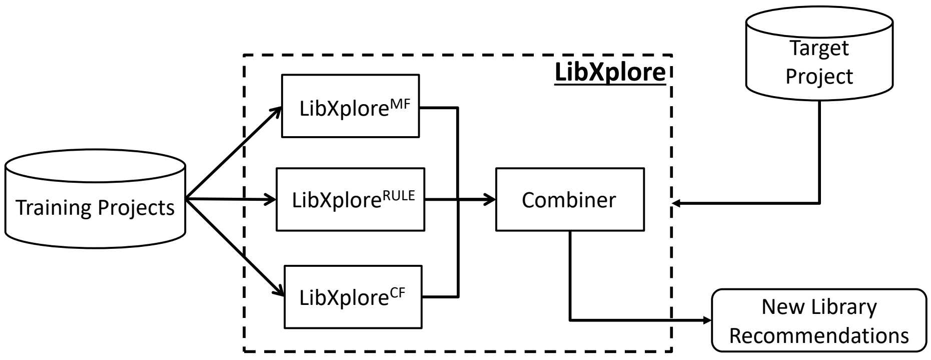


Figure 4.2: Our Library Recommendation Approach $LibXplore$

4.3.2 $LibXplore^{RULE}$ Component

This component is exactly the same as $LibRec^{RULE}$ component from our previous approach (see Section 3.4.2). For the sake of completeness, we briefly describe the details of this component.

Given *Training Projects*, library usage association rules are extracted by following the mining process explained in Section **??**. Given these rules and a *Target Project*, we need to find all rules that match the libraries that are currently used in the project. A library usage rule is considered to match a library if the pre-condition of the rule contains the library. Given a matched rule, we assign likelihood scores to libraries in the post-condition of the rules. The score is simply the confidence score of the matched rule. Note that a library may exist in many matched rules. In such a case, we assign the highest confidence score from the matched rules as the likelihood score of the library. We name the likelihood score given by $LibXplore^{RULE}$ component to library L as $R^{RULE}(L)$. For libraries that do not exist in any of the post-conditions of the matched rules, the corresponding $R^{RULE}(L)$ score is zero.

4.3.3 $LibXplore^{CF}$ Component

This component is similar to $LibRec^{COLLAB}$ component from our previous approach (see Section 3.4.3). The difference lies in how a project is represented. $LibRec^{COLLAB}$ represents projects by the existence of libraries while $LibXplore^{CF}$ represents projects by the textual contents of the libraries that are used in the projects.

$LibXplore^{CF}$ starts by taking all known libraries from *Training Projects*. Given these libraries, $LibXplore^{CF}$ then extracts the names of methods from the call graph of each library. Intuitively, names from method calls in a library should encode information about tasks that a library supports. Adding other kinds of identifiers including variable names might obscure this information. Unlike method name, variable names are not intended to be visible to users and thus might not be properly named.

Given the extracted identifiers, we perform the following text preprocessing:

1. *Tokenization*. Identifiers in method names are often formed from several English words that are concatenated according to a coding standard. In Java, an

identifier name often uses camel casing to concatenate different words. We use a camel case splitter to break down an identifier name to its constituent English words. For example, we break the method name *capitalizeFully* into two words: *capitalize* and *fully*.

2. *Stop word removal*. We remove common English words by using a stop word list.[2] These words occur frequently and thus do not carry much meaning or have much power in differentiating libraries.

3. *Lemmatization*. This process converts an English word to its base form. For example, reading and reads are both converted to read. These kind of words have similar meaning and thus it is typically better to group them. We use Porter Stemmer to perform this process [57].

After preprocessing, the identifiers are considered as a textual document representation for each library. $LibXplore^{CF}$ then generates a vector representation for each library by applying the vector space model (see Section 4.2.2) to the collection of library documents.

These vectors are used to generate project representations. Consider that each project contains a set of libraries that it currently uses and each library now has a vector representation. Let $S = \{V_1, ..., V_n\}$ be a set of library vectors in project P. We define the vector representation of P as follows.

$$V^P = \sum_{i=1}^{n} V_i$$

Consider two library vectors $A = (a_1, a_2, ..., a_m)$ and $B = (b_1, b_2, ..., b_m)$. These vectors have the same size m, which is the number of unique words in all library documents after text preprocessing. The sum of these two vectors is defined as follows.

[2]`http://www.ranks.nl/stopwords`

$$A + B = (a_1 + b_1, a_2 + b_2, ..., a_m + b_m)$$

The above project representation is generated for all projects in *Training Projects* and the *Target Project*. Given the representations, $\mathit{LibXplore}^{CF}$ then calculates cosine similarities between the *Target Project* vector and the vectors of $\mathit{TrainingProjects}$. $\mathit{TrainingProjects}$ are then ranked based on this similarity. We select the top-n projects having the highest similarity with the *Target Project*. In the case that some projects have the same similarity score, we randomly break the tie.

$\mathit{LibXplore}^{CF}$ proceeds to compute a likelihood score for each library in the top-n projects. We collect all of the libraries in the top-n projects and calculates their number of occurrences. Given a library L, we compute the likelihood score for L as follows:

$$R^{CF}(L) = \frac{N_{Lib}(L)}{n}$$

$N_{Lib}(A)$ is the number of top-n projects that use library L. For libraries that do not exist in top-n projects, the corresponding $R^{CF}(L)$ score is zero.

4.3.4 $\mathit{LibXplore}^{MF}$ Component

This component takes as input $\mathit{TrainingProjects}$ and a *Target Project*. Given these projects, $\mathit{LibXplore}^{MF}$ constructs a $m \times n$ project-library usage matrix M where m is the total number of projects and n is the total number of libraries. Values of M are defined as follows.

$$M_{ij} = \begin{cases} 1, & \text{if the } i^{th} \text{ project uses the } j^{th} \text{ library} \\ 0, & \text{otherwise} \end{cases}$$

Given the filled matrix M, we want to factorize this matrix to two matrices U and V:

$$M \simeq UV$$

U is a $m \times f$ project-feature matrix while V is a $f \times n$ library-feature matrix. f is the number of latent features. Thus, our goal is to learn latent features for projects and libraries that result to the observed library usage in M. These latent features encode numerically estimated factors that influence the suitability of a particular library for a particular project.

In this work, we estimate the likelihood of a library to be suitable for a project by taking a dot product between their representative vectors. For project vector $P = (p_1, p_2, ..., p_f)$ and library vector $L = (l_1, l_2, ..., l_f)$, the dot product $(\cdot)$ between P and L is defined as follows.

$$P \cdot L = p_1 \times l_1 + p_2 \times l_2 + ... + p_f \times l_f$$

For example, consider a project having the following latent feature representation: (0.9, 0.2, 0.4). Consider also two libraries having the following latent feature representations: (0.8, 0.2, 0.5) and (0.5, 0.3, 0.5). The dot product between the project vector with the first library vector results in a higher likelihood value. Thus, the first library is more suitable for the project.

We adopt a variant of matrix factorization called non-negative matrix factorization [32]. The objective function for non-negative matrix factorization is used to find U and V by minimizing the difference between M and UV. It is generally defined as follows.

$$\min_{U,V} f(U,V) = \sum_{i=1}^{m}\sum_{j=1}^{n}(M_{ij} - (UV)_{ij})^2$$

$$\text{subject to } U_{ia} \geq 0, V_{bj} \geq 0, \forall i,j,a,b$$

By performing the above optimization, we learn U and V. We can then reconstruct M by multiplying U and V. In this reconstructed matrix, M_{ij} represents the likelihood of the i^{th} project to use the j^{th} library. We refer to the likelihood score from reconstructed matrix M for library L as $R^{MF}(L)$.

4.3.5 *Combiner* Component

This component combines R^{RULE}, R^{CF}, and R^{MF} to generate a final recommendation score, which is denoted as R. The score of library L is defined as follows:

$$R(L) = \alpha \times R^{RULE}(L) + \beta \times R^{CF}(L) + \gamma \times R^{MF}(L)$$

α, β, and γ are the weights for scores from the three ***LibXplore*** components. By default, these weights are set to 1.0.

4.4 Experiments & Analysis

In this section, we present the dataset for our experiments, followed by describing evaluation measures and experimental settings. Next, we list our research questions and describe the experiment results. We end this section by discussing threats to validity.

4.4.1 Dataset

Our dataset is based on the one we use in our previous work [77]. This dataset includes 500 projects that were downloaded from GitHub and has the following properties.

1. *The project has more than 10,000 lines of code.* This is to ensure that the project is real and not a "toy" project.

2. *The project is not forked from another project.* if a project is forked from another, it shares the same source code at the forking time. These original project and its forked projects are likely to share the same libraries. If an original project appears in the training data, while its forked project appears in the test data, the evaluation would be biased.

3. *The project uses Maven.* A project using Maven can be identified by the existence of *pom.xml* file in its root directory. *pom.xml* file specifies project dependencies (i.e., libraries used by the project) that can be uniquely tracked across different projects (see Section 4.2.1).

4. *The project uses at least ten libraries.* The dataset is focused on library intensive projects, from which historical library usages can be mined.

We filter the above dataset further by downloading the libraries and removing the ones that are no longer downloadable. This usually happens because the link is no longer available. Some libraries change name. To download libraries that have changed name, we use information in MvnRepository.[3] This website indexes 205 common Maven repositories (the largest of which is Maven Central) and it also tracks library name change. MvnRepository explicitly states if a library has changed its name and provides a link to a page containing information on where to download the library. We downloaded the library following this information.

[3] `https://mvnrepository.com/`

We split the set of libraries used in the current version of each project into two sets: a set of initial libraries and a set of historically added libraries. The set of libraries used by the current version is identified by analyzing the *pom.xml* file in the latest version of the project. In our previous work, half of the current libraries were dropped and considered as the set of historically added libraries, while the remaining half were considered as the set of initial libraries.

To identify the set of initial libraries among the current libraries, we mine the version control repository for each project. We check out the first project version that contains the initial version of the *pom.xml* file. We then extract the libraries defined in this initial *pom.xml*. We compare it with the set of current libraries. We consider the intersection between the extracted libraries and the set of current libraries as the set of initial libraries. Note that, for some cases, there are other libraries in the extracted libraries that do not exist in the current libraries. We remove such libraries as the scope of our work is to recommend new libraries and not to recommend removal of libraries. To get the set of historically added libraries, We perform a set difference between the set of current libraries and the set of initial libraries. In some cases, the set of extracted libraries and the set of current libraries are exactly the same. We remove such cases from the dataset. Due to this removal, we are left with 486 projects. In our experiment, we use this dataset.

4.4.2 Evaluation Measures and Experimental Settings

For experiments, we use Hit@N as the evaluation measure [53, 54, 66, 71, 82]. Consider that we have m projects for which we want to give library recommendations. Hit@N of a library recommendation system is then defined as the proportion of the m projects for which we can successfully give correct recommendations in the top-N of the returned library recommendation list. Note that Hit@N is the same measure as recall-rate@k that we use in our previous library recommendation work.

To evaluate the effectiveness of our approach, we perform 10-fold cross validation. In 10-fold cross validation, the dataset is randomly split to ten roughly equal sized parts. In each fold, we pick a part and treat it as testing, while the remaining nine parts are treated as training. In the testing data, we consider the set of initial libraries as known libraries in the project and the set of historically added libraries as the ground truth. If a recommended library exists in this ground truth, it means that the recommendation is correct.

LibXplore has several parameters that can be configured: $minsup$ (i.e., the minimum support threshold), $minconf$ (i.e., the minimum confidence threshold), n (i.e., the number of most similar projects), f (i.e., the number of latent factors), α (i.e., the weight of R^{RULE}), β (i.e., the weight of R^{CF}), and γ (i.e., the weight of R^{MF}). By default, we set $minsup$, $minconf$, n, f, α, β, and γ as 0.1, 0.0, 5, 3, 1, 1, and 1, respectively. In our experiments, we also vary these default values and investigate the impact on the effectiveness of our approach.

4.4.3 Research Questions

We want to answer the following four research questions:

RQ1 What is the effectiveness of our approach in recommending libraries to new projects?

RQ2 What is the contribution of each component in our approach to its effectiveness?

RQ3 How does the various components and parameters of our approach affect its effectiveness?

RQ4 How does the various experimental settings affect the effectiveness of our approach?

4.4.4 RQ1: Effectiveness of the Proposed Approach

We have tested our approach on the 486 projects. Using a 10-fold cross validation, we have compared our approach with *LibRec*, a baseline and the state-of-the-art approach (see Chapter 3). We run *LibRec* using its default parameters. The experiment shows that our approach outperforms *LibRec*. Table 4.1 shows that our approach can achieve Hit@5 of 0.584 and Hit@10 of 0.702, which is a 20.44% and 9.69% improvement over *LibRec*. Note that the Hit@5 and Hit@10 scores are conservative estimates to the actual accuracy. This is the case since we consider a library to be correct only if it is later added to a project in the actual historical data. It is possible that other top-ranked libraries are also useful for the project and the original developers were not aware of them.

We notice that the effectiveness of *LibRec* is not as good as the one reported in our previous work. We believe this is due to the more realistic scenario that we now consider in evaluating the approaches – see Section 4.4.1. Our more realistic scenario has cases where the ground truth has a size that is less than half the size of current libraries in a project, which make it more difficult to provide good recommendations. Also, for some projects, the size of the set of ground truth libraries is more than half of that of the set of current libraries in the project. In either case, the task of recommending libraries is more difficult. For the first case, information for inferring good recommendations might be limited. For the second case, there are fewer available number of correct recommendations.

Table 4.1: Effectiveness of the Proposed Approach

Approach	**Hit@5**	**Hit@10**
LibRec	0.504	0.640
LibXplore	0.607	0.702

4.4.5 RQ2: Contributions of Each Component

We want to investigate the contribution of each component in our approach. To do so, we disable each component one at a time and observe its effect on the effectiveness of our approach. To disable $LibXplore^{RULE}$, we set α to zero and investigate the effectiveness of combining $LibXplore^{CF}$+$LibXplore^{MF}$. The performance

difference between this combination and $LibXplore$ indicates the contribution of $LibXplore^{RULE}$. Similarly, setting β to zero would disable $LibXplore^{CF}$ and its contribution can be measured from $LibXplore^{RULE}$+$Lib-$
$Xplore^{MF}$ performance. To measure $LibXplore^{MF}$ contribution, we set γ to zero and check the performance of $LibXplore^{RULE}$+$LibXplore^{CF}$.

Table 4.2 shows the effectiveness of different combinations of components after disabling one of the $LibXplore$ components. Removing $LibXplore^{RULE}$ results in the largest reduction in effectiveness, followed by $LibXplore^{MF}$ and $LibXplore^{CF}$. Removing any of them clearly reduces the effectiveness of $Lib-$
$Xplore$. This shows the benefit of combining the three components – none of the components does not contribute or contributes negatively to the performance of $LibXplore$.

Table 4.2: Effectiveness of Different Component Combinations

Combination	**Hit@5**
$LibXplore^{RULE}$+$LibXplore^{CF}$	0.541
$LibXplore^{RULE}$+$LibXplore^{MF}$	0.592
$LibXplore^{CF}$+$LibXplore^{MF}$	0.520

4.4.6 RQ3: Effectiveness of Various Components and Parameter Settings

We investigate the effectiveness of components of *LibXplore* when they are used separately. We also investigate the effectiveness of our approach when we change the parameter settings. This provides some insights on how the parameters should be changed in practice.

4.4.6.1 Effectiveness of the Individual Components.

We investigate how well $LibXplore^{RULE}$, $LibXplore^{CF}$, and $LibXplore^{MF}$ work individually. The result is shown in Table 4.3. $LibXplore^{MF}$ performs the best, followed by $LibXplore^{CF}$ and $LibXplore^{RULE}$. Compared to Table **??**, combining a component with another generally improves its effectiveness, except when we combine $LibXplore^{MF}$ and $LibXplore^{CF}$. Still, after adding $LibXplore^{RULE}$ to the combination of $LibXplore^{MF}$ and $LibXplore^{CF}$, the effectiveness improves significantly. Again, this shows the benefit of combining the three components together as its effectiveness consistently improves over the effectiveness of its constituent components.

Table 4.3: Effectiveness of Individual Components

Component	**Hit@5**
$LibXplore^{RULE}$	0.420
$LibXplore^{CF}$	0.471
$LibXplore^{MF}$	0.570

4.4.6.2 Effect of Varying Number of Similar Projects

We investigate the impact of considering different numbers of similar projects. Table 4.4 shows the effectiveness of *LibXplore* when we change the number of similar projects in $LibXplore^{CF}$ from 5 to 25. Generally, the effectiveness of our approach reduces when we increase the number of similar projects. Indeed, if this number is too high, some projects may not be that similar and this hurts the performance of $LibXplore^{CF}$.

Table 4.4: Effect of Varying the Number of Similar Projects

Number of Similar Projects	**Hit@5**
5	0.607
10	0.582
15	0.576
20	0.584
25	0.572

4.4.6.3 Effect of Varying *minsup* and *minconf*

We investigate the effect of varying *minsup* in $LibXplore^{RULE}$. As shown in Table 4.5, a higher value of *minsup* generally reduces the effectiveness of our approach. It indicates that some important rules are missed. However, the performance reductions are not very significant as the other components appear to be able to compensate for some of the missing rules.

Table 4.5: Effect of Varying *minsup*

minsup	**Hit@5**
0.1	0.607
0.2	0.592
0.3	0.584
0.4	0.601
0.5	0.592

We also investigate the effect of varying *minconf* in $LibXplore^{CF}$. As shown in Table 4.6, the effectiveness of our approach generally reduces when we increase *minconf*. Similar to increasing *minsup*, increasing *minconf* would also make our approach to miss some useful rules for calculating the final recommendation score. Other components are generally capable of compensating for this loss.

Table 4.6: Effect of Varying *minconf*

minconf	**Hit@5**
0.2	0.607
0.4	0.603
0.6	0.603
0.8	0.566
1.0	0.592

4.4.6.4 Effect of Varying *f*

We investigate the effect of varying *f* in $LibXplore^{MF}$ and show the result in Table 4.7. Increasing *f* generally reduces the effectiveness of our approach. However, the reduction appears not to be very significant. A Higher value of f means a higher number of parameters. Estimating higher number of parameters is generally more difficult as it often involves an increased risk of overfitting.

Table 4.7: Effect of Varying *f* in $LibXplore^{MF}$

f	**Hit@5**
2	0.605
3	0.607
5	0.598
7	0.592
10	0.599

4.4.7 RQ4: Effectiveness for Various Experimental Settings

We investigate the sensitivity of *LibXplore* to two experimental settings: the size of the training set and the number of recommendations.

4.4.7.1 Effect of Varying Training Set Size

To see the effect of training set size, we change the value of k in k-fold cross validation. We show the results in Table 4.8. Higher k indicates a larger training set size. As expected, a larger training size translates to a higher effectiveness of our approach. A smaller training set contains less information, which may not be enough for making effective recommendations.

Table 4.8: Effect of Varying Training Set Size

k	**Hit@5**
2	0.556
4	0.589
6	0.607
8	0.593
10	0.607

4.4.7.2 Effect of Changing Recommendation Size

Here, we investigate the effect of changing the number of libraries recommended by changing the value of N in Hit@N. Table 4.9 shows that our approach achieves an effectiveness of 0.702 when N is 10. In this case, our approach gives, on average, correct recommendations for 70.2% of projects in testing data. However, bringing the correct recommendation to the first rank of returned recommendation list proves to be difficult as only 21% of projects in testing data receive correct recommendations.

Table 4.9: Effect of Varying Recommendation Size

N	**Hit@N**
10	0.702
7	0.650
5	0.607
3	0.459
1	0.210

4.4.8 Threats to Validity

Threats to validity consist of threats to internal validity, threats to external validity, and threats to construct validity.

4.4.8.1 Threats to Internal Validity

Threats to internal validity are related to possible errors in the experiments. To reduce these threats, we have checked our implementations and made sure, to the best of our ability, that they are free from errors. We have made our evaluation setting more realistic, as compared to our previous work.

4.4.8.2 Threats to External Validity

Threats to external validity are related to the generalizability of our experimental results. Our dataset contains only open source Java projects that use Maven. Maven is a popular project management tool that has seen widespread use. We expect many Java projects use Maven. Furthermore, our dataset contains only projects that are hosted in GitHub. However, GitHub is the largest open source software repository in existence today. We have tried to reduce the threats to external validity by experimenting on 486 projects. In the future, we can add more projects to further validate the generalizability of the result.

4.4.8.3 Threats to Construct Validity

Threats to construct validity are related to the appropriateness of our evaluation metric. We evaluate the effectiveness of our approach using Hit@N. This measure is well known and has been used in a number of past studies, e.g., [53, 71, 82]. Thus, we believe that threats to construct validity are minimal.

4.5 Conclusion

We propose a new library recommendation approach that combines association rule mining, neighborhood based collaborative filtering, and non-negative matrix factorization. Our approach leverages information from historical library usage and content of libraries. We have evaluated our approach on 486 open source Java projects hosted on GitHub. Our approach achieves Hit@5 of 0.607 and Hit@10 of 0.702, improving over the state-of-the-art approach by 20.44% and 9.69%, respectively. We introduced a more realistic scenario for evaluating a library recommendation approach. In this scenario, we mined version control repository to split the set of current libraries to the set of initial libraries and the set of historically added libraries. Using these sets, we simulate whether the library recommendation approach can recommend relevant libraries in historically added libraries when given initial libraries as input.

In the future, we plan to increase the size of our dataset to validate generalizability of our approach. We will download more Maven projects from GitHub. We will also download more libraries from Maven repositories. We also plan to further improve the effectiveness of our approach. A potential improvement is by considering textual features from projects, such as project description. Textual features from projects may contain information that explain why a project uses a particular library and not the others. Using these textual features also allows us to recommend libraries to a project that has not used any library at all. It is also likely to significantly improve recommendations for projects that use only a small number of libraries.

Another interesting direction is to integrate our approach in an IDE so that the library recommendation can be performed seamlessly during development. Developers can develop their project as usual and the library recommendation system would notify them if the system finds a library that might be useful for them. Last but not least, we plan to develop an approach that can explain to software developers why a particular library would be useful for their project.

Chapter 5

Automatic Recommendation of API Methods from Feature Requests

5.1 Introduction

Developers often receive requests for new features submitted via systems such as JIRA[1]. Given the requirements expressed in these feature requests, developers need to locate code units that should be changed and then implement the required changes. While a number of concern localization techniques have been proposed for locating code units of interest [16, 58, 60, 80, 90, 92], there is still little automated support to help developers implement the changes required to satisfy a feature request.

Many software systems rely on a variety of external libraries for various functionalities. Accordingly, developers often use external libraries to implement required changes. However, using these libraries effectively, requires knowledge of the relevant methods and classes that they provide. Given the large number of libraries, and the large number of methods and classes that they provide, it can be a challenge for developers to identify the methods and classes of interest, given a target requirement document expressed as a feature request.

Considering the above issues and opportunities, there is a need for an automated

[1]`https://www.atlassian.com/software/jira`

approach that could help developers to better harness the power of libraries. The automated approach should be able to recommend library methods given a feature request. We refer to our problem as *method recommendation from feature requests*.

A number of techniques have been proposed to recommend code units given a requirement. Mandelin et al. [44] and Thummalapenta and Xie [75] propose a technique to generate code snippets that can convert an object of a particular type to another object of a different type. While this technique is useful for a number of situations, it requires the information about the desired functionality to be expressed at code level. Chan et al. propose a code search technique that takes in text phrases and returns a graph of API methods that best match the phrases [13]. Their approach requires *precise* text phrases that match some words in the API methods. These techniques are not sufficient to automatically process feature requests, which typically describe high-level requirements, written in natural language. In this work, we propose a complementary approach that recommends relevant library methods directly from feature requests.

Our proposed approach learns from a training dataset of changes made to a software system recorded in repositories (i.e., issue management systems, and version control systems). Each change in the dataset has three parts: the textual description describing the change (text), the code before the change (pre-change), and the code after the change (post-change). Our approach takes as input a new textual description (text) and then recommends methods from a set of libraries to be used in the post-change code.

To recover methods that can be used to construct the post-change code, our approach performs a two-pronged approach to rank relevant methods. First, it searches for similar *closed* or *resolved* feature requests in the training data. A *closed* or *resolved* feature request is one that has been addressed by developers and where appropriate changes have been made to the software system. It then looks into the API methods that are used to implement these feature requests and measures the relevance of various methods based on the number of similar closed requests which use them. Second, our approach measures an API method relevance by looking into the similarity between the textual description of the feature request and the descriptions of the API method. Our approach then learns an integrated ranking function that is used to recover a list of potentially relevant library methods that are then recommended to the developers.

We have evaluated our solution on feature requests stored in the JIRA issue management systems of 5 Java applications: Axis2/Java, CXF, Hadoop Common, HBase, and Struts 2. Each feature request in JIRA can be linked to the commits in the corresponding version control system that implement the requested feature. We recommend methods from 10 third party libraries: commons-codec, commons-io, commons-lang, commons-logging, junit, servlet-api, easymock, log4j, slf4j-api, and slf4j-log4j12. These are the most popular libraries used by Java applications developed under the Apache Foundation. These libraries provide various functionalities including testing, logging, I/O, etc. The accuracy of our proposed approach is measured using recall-rate@5 and recall-rate@10; these measures have also been used to evaluate past studies on bug report analysis [27, 53, 72, 82]. Our experiments show that we can achieve a recall-rate@5 and recall-rate@10 of 0.690 and 0.779 respectively. On the other hand, we show that the state-of-the-art code search approach by Chan et al. [13] that recommends API methods from precise textual phrases is not effective to directly process feature requests, which often contain high level requirements. Indeed, we find that their approach returns no relevant methods.

Our contributions are as follows:

1. We propose a new problem of *method recommendation from feature requests*.

2. We propose a technique that leverages information from past similar closed or resolved feature requests and compares the textual description of a feature request with those of library methods. Our technique learns an integrated ranking function that is then used to recommend library methods to be used

in the post-change code.

3. We evaluate our approach on change requests of 5 applications and recommend methods from 10 libraries. We show that our approach achieves a recall-rate@5 and recall-rate@10 of 0.690 and 0.779, respectively.

The structure of this chapter is as follows. In Section 5.2, we describe some preliminary concepts. In Section 5.3, we present an overview of our proposed approach. We elaborate the three processing components of our approach in Sections 5.4, 5.5, and 5.6. We highlight our experimental methodology and results in Section 5.7. Finally, we conclude in Section 5.8.

5.2 Preliminaries

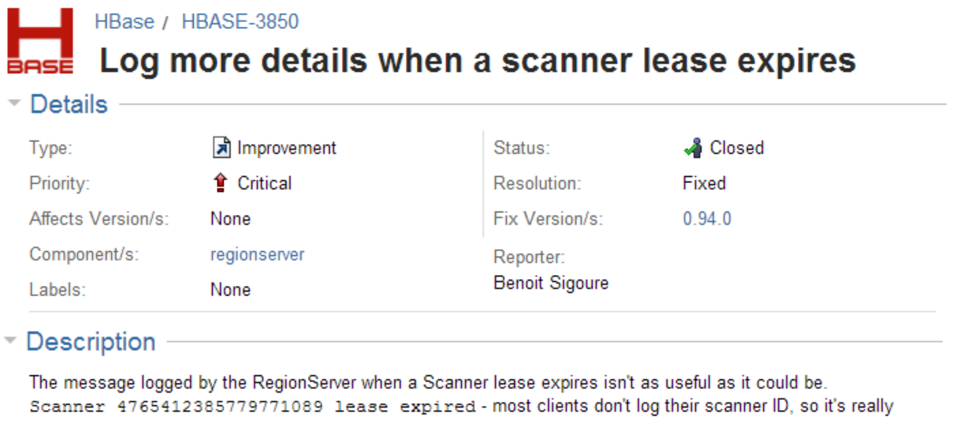
In this section, we describe some preliminary materials that are needed for later sections. We first describe the issue management system JIRA and show how it stores feature requests. We then describe some text pre-processing techniques and the vector space model.

5.2.1 Feature Requests and JIRA

JIRA is an issue management system developed by Altassian.[2] It is used in many software projects to capture and store issues that are reported by users and developers. Among its users are the many projects developed by the Apache Software Foundation. Figure 5.1 shows a sample issue stored in the JIRA repository of an Apache project. An issue contains a number of fields including: summary, description, type, priority, component, etc. For our work, we are especially interested in the fields listed in Table 5.1.

Each issue in JIRA can be categorized into one of these types: “Bug”, “New Feature”, “Task”, etc. In this study, we are interested in feature requests reported in

[2]http://www.atlassian.com/software/jira/overview


HBase / HBASE-3850

Log more details when a scanner lease expires

Details

Type:	Improvement	Status:	Closed
Priority:	Critical	Resolution:	Fixed
Affects Version/s:	None	Fix Version/s:	0.94.0
Component/s:	regionserver	Reporter:	Benoit Sigoure
Labels:	None		

Description

The message logged by the RegionServer when a Scanner lease expires isn't as useful as it could be.
`Scanner 4765412385779771089 lease expired` - most clients don't log their scanner ID, so it's really

Figure 5.1: A Sample JIRA Issue

Table 5.1: Fields in a JIRA Issue

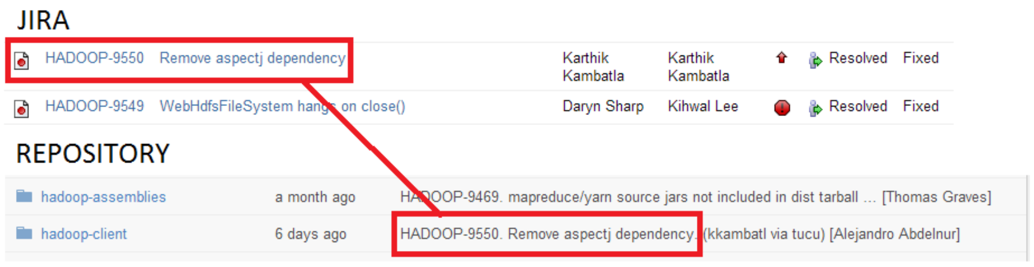
Name	**Description**
Summary	the summary/title of the issue
Description	the detailed description of the issue
Component	the component affected by the issue
Reporter	the name of the person who submitted the issue report
Priority	the urgency of the issue to be addressed

JIRA. A feature request in JIRA can be seen as an issue of type: “New Feature”, “Improvement”, or “Wish”. Each issue also has a priority. The priority indicates the urgency of the issue to be addressed. Table 5.2 lists the priorities available in JIRA along with their descriptions.

Table 5.2: Priority in JIRA

Name	**Description**
Blocker	Blocks development and/or testing, production could not run
Critical	Crash, loss of data, severe memory leak
Major	Major loss of functionality
Minor	Minor loss of functionality, or other problem where an easy workaround is present
Trivial	Cosmetic problem like misspelled words or misaligned text

An issue can be assigned various status labels: “open”, “in progress”, “resolved”, “closed”, etc. A new issue is typically given the status “open”. An issue that has been addressed to completion by developers is given the status: “resolved” or “closed”. Each issue report in JIRA has a unique issue identifier (id) used to identify the report. The format of this identifier is typically a short name for the project followed by the issue number in the project (e.g., HBASE-3850). JIRA can be integrated with version control systems like svn, git, etc. Each issue in JIRA can then be linked to the commits in the version control system that address the issue. The issue identifier is added to the log messages of the commits that address the issue. This provides an easy identification of changes made to address the issue. We show an example of this link in Figure 5.2. We can see that it is easy to identify the commits in the version control system that address the *HBASE-3850* issue.


JIRA

HADOOP-9550	Remove aspectj dependency	Karthik Kambatla	Karthik Kambatla	Resolved	Fixed
HADOOP-9549	WebHdfsFileSystem hangs on close()	Daryn Sharp	Kihwal Lee	Resolved	Fixed

REPOSITORY

hadoop-assemblies	a month ago	HADOOP-9469. mapreduce/yarn source jars not included in dist tarball ... [Thomas Graves]
hadoop-client	6 days ago	HADOOP-9550. Remove aspectj dependency. (kkambatl via tucu) [Alejandro Abdelnur]

Figure 5.2: Sample Link Between A JIRA Issue And A Commit in A Version Control System

5.2.2 Text Pre-processing

Text pre-processing is an important task in text mining [45]. Its purpose is to convert a piece of text into a common representation easily processed by a text mining algorithm and to remove certain noise. Widely used text pre-processing strategies include tokenization and stemming.

Tokenization refers to the process that breaks a document into word tokens. Delimiters are used to demarcate one token from another. Typically, space and punctuation are used as delimiters. After tokenization, a document is converted to a bag (i.e., a multiset) of word tokens. This is often referred to as the *bag of words* representation.

Stemming is the process of converting a word to its base form. This base form is usually called the stem word. For example, word “argue”, “argues”, “argued”, and “arguing” have a common stem word “argu”. Even though word “argu” is not a dictionary word, the conversion assures that we can identify a word in its different forms and link these word forms together. Without stemming, the multiple forms would be treated as different words altogether, which is not desirable in many cases. In our work, we use the Porter stemmer[3] to stem the words. It employs several rule based heuristics to convert a word to its stem word by stripping a suffix of the word. The Porter stemmer has been used in many past software engineering studies, e.g., [21, 25].

5.2.3 Vector Space Model

After the text pre-processing step, the document is now represented as a bag of words. The vector space model represents a bag of words as a vector of weights. Each word in the bag becomes an element in the vector. The weight of each word indicates its importance. Term frequency and inverse document frequency are often used to compute the weight of a word and thus quantify its importance in a document.

Term frequency (TF) refers to the number of times a term (i.e., a word, or a token) appears in a document. The more times a term appears in a document, the more important that term is considered to be. Inverse document frequency (IDF) is the reciprocal of the document frequency (DF). The document frequency of a term is the number of documents in the corpus (i.e., a set of documents or a document collection under consideration, e.g., all feature requests, all method descriptions in the API documentation) that contain the term. The higher the inverse document frequency is, the more important is the term, as it can better differentiate one document from another. TF-IDF is often used to compute the weight of a term i in a document D considering a corpus C in the following way:

$$\begin{aligned} w_{i,D,C} &= TF_{i,D} \times IDF_{i,C} \\ IDF_{i,C} &= \tfrac{1}{DF_{i,C}} \end{aligned} \tag{5.1}$$

[3]`http://tartarus.org/martin/PorterStemmer/`

$TF_{i,D}$ refers to the number of times a word i appears in a document D. $DF_{i,C}$ refers to the number of documents in C that contains the word i.

From the above, given a bag of words representing a document D in a corpus C, we can convert it to its corresponding term vector by computing the weight of each word in C and putting them into a vector. The weight of a word in C, but not in D, would be 0. We denote the term vector representation of document D considering a corpus C as $VSM_C(D)$. Implementation-wise, a sparse vector representation is typically used (i.e., only the non-zero entries of the vector are stored).

Given two documents, we can compute the similarity between them by comparing their representative vectors. Cosine similarity is often used to measure the similarity between two vectors. Let V_1 and V_2 denotes two vectors of weights of size N, then the cosine similarity of these two vectors is given by the following equation:

$$Cosine(V_1, V_2) = \frac{\sum_{i=1}^{N} w_{i,V_1} \times w_{i,V_2}}{\sqrt{\sum_{i=1}^{N} w_{i,V_1}^2}\sqrt{\sum_{i=1}^{N} w_{i,V_2}^2}} \tag{5.2}$$

$w_{i,V}$ refers to the i^{th} weight in vector V.

5.3 Overall Framework

The overall framework of our approach is shown in Figure 5.3. Our framework consists of three important components: *History Based Recommender*, *Description Based Recommender*, and *Integrator*.

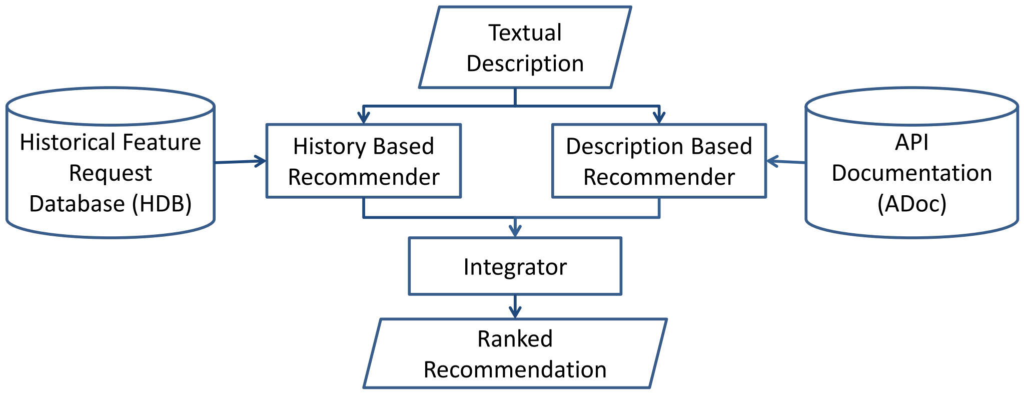

Textual Description
Historical Feature Request Database (HDB)
History Based Recommender
Description Based Recommender
API Documentation (ADoc)
Integrator
Ranked Recommendation

Figure 5.3: Our Recommendation Framework

History Based Recommender takes as input the description of the new feature request (*Textual Description*) and a historical database containing old "closed" or "resolved" feature requests (*Historical Feature Request Database (HDB)*). The recommender compares the new feature request with those in the historical database and finds the closest ones. It then recommends relevant methods based on the methods that were used to implement those closest feature requests.

Description Based Recommender takes as input the description of the new feature request (*Textual Description*) and the documentation of API libraries (*API Documentations (ADoc)*). The recommender computes the similarity of the textual description of the new feature request with the description of each method in the API documentations of the libraries. It recommends methods whose textual descriptions have the highest similarity with the textual description of the new feature request.

Integrator combines the recommendations from *History Based Recommender* and *Description Based Recommender*. It takes as inputs recommendation scores from the two components and outputs a final list of methods to be recommended to the user.

5.4 History-Based Recommendation

In the history-based recommender component, we first find the nearest neighbors of a new feature request from the historical database of "closed" or "resolved" feature requests that we have. We compare two feature requests based on the contents of their summary, description, component, reporter, and priority fields (see Table 5.1). We compute a similarity score for each field and combine the scores into an aggregate score that specifies the similarity between two feature requests. We define the similarity score for each field as follows.

1. ***Summary* and *Description***. The contents of these fields are free-form texts. We pick only alphanumeric terms from these texts. We employ standard text preprocessing (tokenization and stemming) and convert the terms into a bag

of word and its corresponding term vector (see Section 5.2). We have 3 options: we can take all terms in the summary field, we can take all terms in the description field, and we can take all terms in both summary and description fields. We compute the following 3 similarity scores between the new feature request F_1 and a historical feature request F_2 in terms of their summary/description fields using cosine similarity:

$$\begin{aligned} &SimSum(F_1, F_2) = Cosine(VSM_{HDB}(F_1^S), VSM_{HDB}(F_2^S)) \\ &SimDesc(F_1, F_2) = Cosine(VSM_{HDB}(F_1^D), VSM_{HDB}(F_2^D)) \\ &SimSumDesc(F_1, F_2) = Cosine(VSM_{HDB}(F_1^{SD}), VSM_{HDB}(F_2^{SD})) \end{aligned} \tag{5.3}$$

In the above equations, F_1^S denotes the content of the summary field of F_1. F_1^D denotes the content of the description field of F_1. F_1^{SD} denotes the contents of the summary and description fields of F_1. HDB is the Historical Feature Request Database (see Figure 5.3).

2. ***Component***. A feature request, if implemented, can affect multiple components in the system. Thus, the content of the component field of a feature request is a set of values. We compute the similarity score $SimComp$ between a new feature request $F1$ and a historical feature request $F2$ in terms of their components as follows:

$$SimComp(F_1, F_2) = \frac{|Nc_{F_1} \cap Nc_{F_2}|}{\sqrt{|Nc_{F_1}|} * \sqrt{|Nc_{F_2}|}}$$

Nc_F denotes the set of components of feature request F.

3. ***Reporter***. The similarity score $SimReport$ between a new feature request F_1 and a historical feature request F_2 in terms of their reporters is 1 if both of them have the same reporter and 0 otherwise.

4. ***Priority***. Each priority in JIRA can be assigned an ordinal value to quantify its level of urgency. We assign value 1 for “Blocker”, 2 for “Critical”, 3 for

"Major", 4 for "Minor", and 5 for "Trivial". A lower value means a higher priority or level of urgency. We compute the similarity score $SimPrio$ between a new feature request F_1 and a historical feature request F_2 in terms of their priority based on these values. The formula is as follows:

$$SimPrio(F_1, F_2) = \frac{1}{1 + |Prio_{F_1} - Prio_{F_2}|}$$

$Prio_F$ denotes the ordinal value corresponding to the priority of feature request F.

Example. Consider the example feature request shown in Figure 5.1 as a historical feature request and a new feature request having values as shown in Table 5.3. We can compute the similarity between these two feature requests for each field as follows.

1. ***Summary* and *Description***. Since the computation steps for both summary and description are basically the same, in this example, we only compute the similarity score for the summary. We convert a summary to a vector of $TF - IDF$ weights of its stemmed alphanumerical words. Each word has a term frequency TF equal to 1. Assuming that the IDF of each word is 1, which means the word only appears in one document in the historical database of feature requests (HDB), the similarity score for the summaries of the historical and new feature request (i.e., $SimSum$) is $1/(\sqrt{8} * \sqrt{6}) = 0.144$.

2. ***Component***. The historical feature request and the new feature request do not share any component. Thus, the $SimComp$ score is 0.

3. ***Reporter***. The feature requests are reported by different reporters so the $SimReport$ score is 0.

4. ***Priority***. The historical feature request has "Critical" priority which corresponds to the ordinal value 2, while the new feature request has "Minor" pri-

ority which corresponds to the ordinal value 4. The denominator of $SimPrio$ is equal to $1 + |2 - 4| = 3$. Thus, the $SimPrio$ score is $1/3 = 0.333$.

Table 5.3: Example of A New Feature Request

Field	Values
ID	HBASE-6372
Summary	Add scanner batching to Export job
Description	When a single row is too large for the RS heap then an OOME can take out the entire RS. Setting scanner batching in custom scans helps avoiding this scenario, but for the supplied Export job this is not set. Similar to HBASE-3421 we can set the batching to a low number - or if needed make it a command line option.
Components	mapreduce
Reporter	Lars George
Priority	Minor

Finally, to compute the final similarity score between two feature requests, we aggregate the similarity scores of their constituent fields. We compute the final similarity $Sim^{HISTORY}$ between a new feature request F_1 and a historical feature request F_2 in the historical database using the following formula:

$$\begin{aligned}&Sim^{HISTORY}(F_1, F_2) =\\&\alpha_1 * SimSum(F_1, F_2) + \alpha_2 * SimDesc(F_1, F_2) +\\&\alpha_3 * SimSumDesc(F_1, F_2) + \alpha_4 * SimReport(F_1, F_2) +\\&\alpha_5 * SimComp(F_1, F_2) + \alpha_6 * SimPrio(F_1, F_2)\end{aligned} \tag{5.4}$$

$SimSum(F_1, F_2)$, $SimDesc(F_1, F_2)$, $SimSumDesc(F_1, F_2)$, $SimReport(F_1, F_2)$, $SimComp(F_1, F_2)$, and $SimPrio(F_1, F_2)$ denote the similarity scores between F_1's and F_2's summary, description, combination of summary and description, reporter, components, and priority respectively. α_1-α_6 denotes the weights of each field contributing to the $Sim^{HISTORY}$ score.

Given a new feature request, we rank the historical feature requests in the *Historical Feature Request Database* based on their $Sim^{HISTORY}$ scores when compared to the new feature request. The higher the score is, the more similar a historical feature request is to the new feature request. We then pick the top-k feature requests with the highest $Sim^{HISTORY}$ scores. If there are feature requests with rank greater than k that have the same score as the k-th feature request, we group the feature requests having this score. We then randomly select feature requests from this group until we have k nearest neighbors (i.e., ties are randomly broken).

Next, we compute the recommendation scores for each method based on these top-k nearest neighbors. We collect the methods that are used to implement the feature requests in the top-k nearest neighbors and compute a score for each method. Given a method m, the *history based recommendation score* of API method m for feature request F, denoted as $RecScore^{HISTORY}(F, m)$ is computed as follows:

$$RecScore^{HISTORY}(F, m) = \frac{NNCount_{Method}(F, m)}{k} \tag{5.5}$$

$NNCount_{Method}(m)$ denotes the number of nearest neighbors of feature request F that use API method m, and k denotes the total number of nearest neighbors. By default, we set the number of nearest neighbors k to be 5. The API method with the highest $RecScore^{HISTORY}$ score is the most suitable API method based on our history-based recommender.

Example. Consider a top-2 nearest neighbor list containing N_1 and N_2. Feature request N_1 was implemented using method m_1 and m_2 while feature request N_2 was implemented using method m_2. Thus, the value of $NNCount_{Method}$ is 1 for m_1 and 2 for m_2. We can then compute $RecScore^{HISTORY}$ score of m_1 and m_2 which are 0.5 and 1.0 respectively.

5.5 Description-Based Recommendation

When adding a new feature to an application, developers often look at the API documentation to see which methods they can use to help them implement the feature. The API documentation contains textual descriptions that explain each method in the library. By looking at the documentation, developers can find out which API methods to use for implementing a feature. Our description-based recommender component mimics this process to find relevant methods given the textual description of a feature request. Given a new feature request F, it proceeds in the following steps:

1. **Feature Request Preprocessing.** We extract the contents of the summary and description fields of the input feature request F. We again perform standard text preprocessing steps to convert them into a bag of words. This bag of word is then converted into its corresponding term vector representation $VSM_{ADoc}(F)$ where each token (i.e., term) in the bag is represented by its TF-IDF weight[4] and $ADoc$ is API Documentation (see Figure 5.3).

2. **API Method Preprocessing.** For each API method m that we consider, we extract its method signature and its corresponding description in the API documentation. We extract the method descriptions from the Javadoc comments in the code base of the APIs. We make use of Eclipse Java Development Tools (JDT) to extract these Javadoc comments. Javadoc has tags which serve as metadata. Examples include *@param* indicating the start of the description of a method parameter, *@return* indicating the start of the description of the return value of a method, etc. Since these tags only serve as metadata and are not specific to the API, we remove them from the extracted Javadoc comments. Additionally, developers sometimes add HTML tags in the documentation to improve its readability when it is viewed in e.g., a web browser. Since these tags are only meant to improve the look and feel of the API documentation and are again not specific to the API, we also remove all HTML tags. Next, we perform standard text preprocessing (i.e., tokenization and stemming) to convert the cleaned method descriptions in the Javadoc comments into bags of words. We then convert each bag of words into its corresponding term vector representation, $VSM_{ADoc}(m)$.

[4]The description of text preprocessing and vector space model is given in Section 5.2.

3. **Similarity Computation.** Next, for each method m, we compute the similarity between $VSM_{ADoc}(F)$ and $VSM_{ADoc}(m)$. We use the cosine similarity to compute the similarity between these two vectors (see Section 5.2). We refer to this similarity score as the *description based recommendation score* $RecScore^{DESCRIPTION}(F, m)$ between feature request F and API method m:

$$RecScore^{DESCRIPTION}(F, m) = Cosine(VSM_{ADoc}(F), VSM_{ADoc}(m)) \tag{5.6}$$

After the above steps, we have the $RecScore^{DESCRIPTION}$ scores of various API methods. The method with the highest score is the most relevant API method based on the description-based recommender.

5.6 Unifying History- and Description-Based Recommendation

The last component in our framework is the Integrator, which combines the scores from the previous components. We compute the final recommendation score $RecScore$ between feature request F and API method m by combining $RecScore^{HISTORY}(F, m)$ and $RecScore^{DESCRIPTION}(F, m)$, as follows:

$$\begin{aligned} RecScore(F, m) = \alpha * RecScore^{HISTORY}(F, m) + \\ \beta * RecScore^{DESCRIPTION}(F, m) \end{aligned} \tag{5.7}$$

α and β are the weights for $RecScore^{HISTORY}$ and $RecScore^{DESCRIPTION}$, respectively.

To set the appropriate values for α and β of $RecScore$ (see Equation 5.7) and the appropriate values for α_1-α_6 of $RecScore^{HISTORY}$ (see Equation 5.5), we heuristically find the best set of weights that maximizes an evaluation metric based on a training dataset We employ a greedy approach based on Gibbs sampling [12] that iteratively refines the set of weights. At each iteration, each weight is optimized independently. Several iterations are performed to further optimize the weights. The pseudocode of our approach to tune the set of weights is shown in Figure 5.4.

Our algorithm takes the input historical feature requests, a set of API documentations, and the number of iterations to perform Gibbs sampling $numIter$. It then outputs the set of best weights. Initially all weights ($\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha, \beta$) are set to 1.0 (Line 8). We then iterate $numIter$ times (Lines 11-23). For each iteration, we try to estimate the best $\alpha_1, \alpha_2, \alpha_3,$
$\alpha_4, \alpha_5, \alpha_6, \alpha, \beta$ weights independently (Lines 12-22). We go through each of the eight weights and for each weight we investigate 11 settings (i.e., 0.0, 0.1, 0.2, $\ldots$, 1.0) (Lines 13-21). We pick the setting that give the best result (Lines 16-19,21). Method $eval$ evaluates how good a particular weight setting is with respect to an evaluation criteria (Line 16). In this study, we make use of recall-rate@k [53, 66, 70, 72, 82] as the evaluation criteria (see Section 5.7). At the end of the above process, we would have estimated the best weights.

In the end, we want to get the top-k methods based on $RecScore$. To do this, we first get the set of methods with non-zero $RecScore^{HISTORY}$ scores. For all these methods, we compute their $RecScore$ scores. We then return the top-k methods based on the $RecScore$ scores. If there are methods having the same score as the k-th method, we group the methods having this score and randomly select methods from this group until we have k top methods (i.e., ties are randomly broken).

5.7 Experiments & Analysis

In this section, we first describe our dataset. We then outline our experimental methodology and research questions. Next, we present the answers to the research questions and describe some threats to validity.

```
1: **Input:**
2:    $FReqs$ = list of historical closed or resolved feature requests
3:    $Docs$ = the documentation of APIs
4:    $numIter$ = number of iteration
5: **Output:**
6:    Estimated best weights: $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha, \beta\}$
7: **Method:**
8: **Let** $weights$ = {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}
9: **Let** $maxEvalScore$ = 0
10: **Let** $valForMax$ = 0
11: **for** $i = 0$ to $numIter$ **do**
12:    **for** $j = 0$ to $weights.Length$ **do**
13:       **for** $k = 1.0$ to $0.0$ by $0.1$ **do**
14:          $weights[j] = k$
15:          $evalScore = eval(FReqs, Docs, weights)$
16:          **if** $maxEvalScore > evalScore$ **then**
17:             $maxEvalScore = evalScore$
18:             $valForMax = k$
19:          **end if**
20:       **end for**
21:       $weights[j] = valForMax$
22:    **end for**
23: **end for**
24: **return** $weights$
```

Figure 5.4: Pseudocode for our Weight Tuning Algorithm

5.7.1 Dataset

We first describe how we select libraries of interest and the projects that we investigate. Next, we describe the feature requests that we use to evaluate our approach.

5.7.1.1 Library Selection

We pick libraries that are frequently used by many projects of the Apache Foundation. We choose Apache projects that use Maven as their project management tool. Maven includes a dependency management feature which helps us resolve the libraries used by the projects. These libraries have standard names in Maven, so it is easy to reliably match the libraries that are used across different projects. We first download Apache projects that use Java as their main programming language. We then filter these projects based on the existence of the *pom.xml* file in their root directory. This *pom.xml* file indicates the use of Maven as the project management tool. After this filtering process, we have 207 Apache projects. For each project, we extract the libraries it depends on and count the number of projects using each library. We then rank the libraries based on the number of projects using it. We list the top-10 libraries in Table 5.4. These are the target libraries for our study – we recommend methods from these libraries.

Table 5.4: Top-10 Most Popular Libraries in 207 Apache Projects

Name	**Description**
commons-codec	common encoder and decoder library for string, URL, etc
commons-io	common library for input/output functionality
commons-lang	common library providing extra methods for manipulating Java core classes
commons-logging	common library which encapsulates the logging process for different logging implementations
easymock	a library that provides easy way to use mock objects in unit testing
junit	unit testing framework
log4j	logging library
servlet-api	library providing contracts between a servlet and the runtime environment
slf4j-api	an abstraction library for various logging framework
slf4j-log4j12	a binding library for slf4j and log4j

5.7.1.2 Project Selection

Next, we want to pick large projects ($>$ 100,000 lines of code) from the 207 Apache Foundation projects whose feature requests we use to evaluate our approach. We omit “toy” and small projects. We filter out projects that only use a few of the 10 selected libraries. We choose these projects as we only recommend methods from the 10 libraries. We also filter out projects that do not use JIRA issue management system. We choose these projects as we need reliable links between bug reports and corresponding commits in the version controls system. These links are well maintained in JIRA issue management systems, c.f., [8]. Table 5.5 lists the projects that we use in this study and their descriptions.

Table 5.5: Selected Apache Projects

Name	Description
Axis2/Java	server-client web service engine
CXF	open source web service framework
Hadoop Common	common utilities used in other Hadoop modules
Hbase	scalable distributed database based on Big Table [14]
Struts 2	enterprise-ready web application framework

5.7.1.3 Feature Request Selection and Gold Standard Extraction

We pick feature requests from the JIRA issue management system of the selected projects. We pick only issues in JIRA that are of relevant types. As mentioned in Section 5.2.1, there are 3 issue types in JIRA that correspond to a feature request, namely “New Feature”, “Improvement”, and “Wish”. For these three issue types, we pick issues that are either “closed” or “resolved”.

JIRA contains explicit links between issue reports and repository commits. Using these links, we find the changed files for each commit that addresses an issue. These files have a pre-change and a post-change version. We extract the methods from the libraries shown in Table 5.4 that are invoked in the post-change version of the file as the evaluation benchmark or gold standard (c.f. [45]). A good recommendation system should be able to recommend these methods. There are three cases that we need to consider when extracting method calls for gold standard:

1. *File is added in the post-change version.* If the file does not exist in the pre-change version, we take all the methods from the 10 libraries that are invoked in the post-change version as members of the gold standard.

2. *File is changed in the post-change version.* If the file exists in both pre-change and post-change versions, we take as the gold standard all the methods from the 10 libraries that are invoked in the post-change version, but not invoked in the pre-change version.

3. *File is deleted in the post-change version.* If the file is deleted, the file does

not contribute any API method to the gold standard.

Since we only recommend methods from the top-10 libraries, we only take feature requests whose gold standard set contains at least one method from the 10 libraries. We ignore very rare methods that are only used in one feature request. Our history-based approach requires a method to be used in at least 2 feature requests. Table 5.6 shows the number of feature requests for each of the projects that we use in this study.

Table 5.6: Number of Feature Requests in Our Dataset

Project	**#Feature Request**
Axis2/Java	108
CXF	106
Hadoop Common	79
Hbase	161
Struts 2	53
Total	**507**

5.7.2 Methodology & Research Questions

In order to measure the effectiveness of our approach, we use a commonly used evaluation measure namely *average recall-rate@k* [66]. *Recall-rate@k* has a value of either 1 or 0 where k is the number of the returned items. It has value 1 if at least one of the k returned items (i.e., recommended method) is a member of the gold standard and 0 otherwise. We use *recall-rate@k* as it has also been used in many past studies that also analyze entries in issue management systems [53, 66, 70, 72, 82].

For each project, we perform stratified ten-fold cross validations to evaluate the effectiveness of our approach. We divide the feature requests of a project randomly into ten groups of roughly equal sizes (± 1) and then perform ten iterations. For each iteration, one group is used as the test data (i.e., they form the set of new feature requests) and the remaining nine groups are combined to be the training data (i.e., they form the historical feature request database (HDB)). The test data is used to evaluate the effectiveness of our approach. We report the average effectiveness over the ten iterations.

We consider the following research questions:

RQ1 What is the effectiveness of our proposed approach?

RQ2 What are the relative contributions of the various components of our approach?

RQ3 How effective is our approach compared to a state-of-the-art code search based approach in recommending relevant methods for a feature request?

5.7.3 Experimental Results

We describe the answers to each of the research questions below.

5.7.3.1 RQ1: Effectiveness of the Proposed Approach

Table 5.7 shows the effectiveness of our approach. The average recall-rate@5 and recall-rate@10 are 0.690 and 0.779 respectively. We show that, by only returning 5 methods, our approach can correctly recommend relevant methods for 57.1% to 80.5% of the feature requests in a project. In other words, our approach can put relevant methods in high ranking positions. If we increase the recommendation size to be 10 methods, our approach can correctly recommend at least one relevant method for 70.9% to 90.8% of the feature requests.

5.7.3.2 RQ2: Relative Contributions

Our proposed approach has two main components: the history-based recommender and the description-based recommender. Our history-based recommender computes the similarity between two feature requests by aggregating 6 similarity scores (based on summary, description, summary + description, reporter, component and priority).

Table 5.7: Effectiveness of Our Approach

Project	**Recall-Rate@5**	**Recall-Rate@10**
Axis2/Java	0.805	0.908
CXF	0.628	0.725
Hadoop Common	0.571	0.709
HBase	0.789	0.839
Struts 2	0.657	0.713
Overall	**0.690**	**0.779**

In this research question, we want to investigate the relative contributions of the various components and sub-components of our approach.

We employ Gibbs sampling to tune 8 weights to yield a semi optimal setting. Thus, we can estimate the contributions of the various components and sub-components of our approach from their corresponding weights. The average values of the 8 weights across the ten fold cross validation performed for computing recall-rate@5 and recall-rate@10 are shown in Table 5.8. We find that all components and sub-components of our approach are important as none of them is given a weight of zero. Different weights for different projects indicate different importance of our components for different projects. For both $k = 5$ and $k = 10$, α_1 has the lowest weight compared to the other parameters, suggesting that it is less useful than the other information leveraged by our approach. In our approach, each recommended method needs to have a non-zero $RecScore^{HISTORY}$ score (see Section 5.6). However, this does not mean that the $RecScore^{DESCRIPTION}$ score is not useful. Indeed, we note that the weight of our description-based recommendation score (*Description*) is higher than that of our history-based recommendation score (*Historical*). This indicates that among methods with non-zero $RecScore^{HISTORY}$ score, we can use $RecScore^{DESCRIPTION}$ scores to rank them.

5.7.3.3 RQ3: Comparison with a Code Search Based Approach

Code search can also be used to recommend relevant API methods. Chan et al. propose a graph-based approach that can search an API library for relevant methods

Table 5.8: Average Weights for K = 5 and 10

Project	**k**	**Weight**							
		α_1	α_2	α_3	α_4	α_5	α_6	α	β
Axis2/Java		0.40	0.43	0.59	0.31	0.44	0.70	0.61	0.70
CXF		0.60	0.61	0.72	0.87	0.74	0.82	0.72	0.81
Hadoop Common	@5	0.70	0.61	0.56	0.35	0.63	0.70	0.51	0.60
HBase		0.38	0.29	0.36	0.20	0.23	0.60	0.52	0.70
Struts 2		0.58	0.10	0.49	0.18	0.05	0.61	0.42	0.71
Average		**0.532**	**0.408**	**0.544**	**0.382**	**0.418**	**0.686**	**0.556**	**0.704**
Axis2/Java		0.54	0.59	0.45	0.19	0.27	0.42	0.71	0.80
CXF		0.50	0.51	0.51	0.63	0.54	0.68	0.77	0.60
Hadoop Common	@10	0.52	0.65	0.58	0.51	0.64	0.54	0.52	0.51
HBase		0.35	0.54	0.36	0.31	0.42	0.15	0.10	0.10
Struts 2		0.77	0.62	0.73	0.36	0.79	0.70	0.73	1.00
Average		**0.536**	**0.582**	**0.526**	**0.400**	**0.532**	**0.498**	**0.566**	**0.602**

given a set of text phrases [13]. To the best of our knowledge, this is the closest study to our work. Their approach processes the text queries and returns a connected graph whose nodes are methods. They have evaluated their approach on a set of *precise* text queries that contain keywords that match desired methods. For example, for input queries containing phrases: store, folder, open, and search, they output several relevant methods including: *javax.mail.Store:getDefaultFolder()*, *javax.mail.Folder-:open(int mode)*, etc. Note that the input queries contain keywords that must appear in the signatures of the relevant methods. We want to investigate if their approach can also handle feature requests.

To do this, we preprocess a feature request into text phrases. We treat each word that appears in the summary and description fields of a feature request as a text phrase. We then run their tool on our processed data. Table 5.9 show the average number of methods that are returned in the connected graphs returned by their tool. Even though the tool returns a number of methods, unfortunately none of them are relevant for each of the 507 feature requests (i.e., their recall-rate@5 and recall-rate@10 are both 0). This shows that approaches that process precise text queries cannot handle feature requests. Indeed, feature requests often contain high level requirements while methods contain low level requirements. Our proposed approach tackles this problem by leveraging historical feature requests.

Table 5.9: Average # of Returned Methods by Chan et al.'s Approach

Project	**Average # of Returned Methods**
Axis2/Java	2.2
CXF	1.8
Hadoop Common	1.7
HBase	1.8
Struts 2	1.7
Average	**1.84**

5.7.4 Threats to Validity

Threats to internal validity refers to experimental bias and errors. We have checked our code and data for errors. Still there could be errors that we have not noticed. We also ensure that we do not mix training and test data in our cross validation. For the feature importance measurement, we did not perform feature redundancy analysis and thus it is possible that some of the features are redundant.

Threats to external validity refers to the generalizability of our proposed approach. In this study, to address this threat, we have considered a few hundred feature requests from 5 software systems. We have also recommended methods from 10 libraries. In the future, we plan to reduce this threat further by analyzing more feature requests from additional software systems and recommending methods from more libraries. We have also performed cross validation, which is the standard approach to assess how a proposed approach would perform on an independent dataset.

Threats to construct validity refers to the suitability of our evaluation metrics. We make use of recall-rate@k which is a commonly used metric in many past studies [53, 66, 70, 72, 82]. Thus, we believe there is little threat to construct validity.

5.8 Conclusion

In this work, we have proposed a new method recommendation approach that takes as input a feature request and recommends methods from a set of libraries. In contrast to previous approaches, our approach does not require precise input information, such as precise input or output types, or precise matching textual descriptions. Thus, it is suitable for directly processing feature requests stored in bug repositories, which often do not precisely specify relevant code elements. Our approach is a hybrid approach, combining history-based recommendation and description-based recommendation. On 10 libraries and 507 feature requests from 5 software systems, we achieve an average recall-rate@5 and recall-rate@10 of 0.690 and 0.779 respectively. We have also compared our approach to the latest method recommendation technique that requires precise textual descriptions from end users and show that it is not useful for recommending methods from feature requests.

In the future, we plan to improve our solution further to achieve even higher recall-rate@k scores. Some possible directions include using state-of-the-art natural language processing [10, 81], taking the information stored in the code base into account, and specification mining e.g., [15, 29, 36, 37, 38, 39, 40, 41, 68]. We also plan to experiment with a larger number of feature requests from more software systems and to perform a more thorough investigation of the factors that affect the effectiveness of the different components of our approach in contributing towards the effectiveness of the proposed solution. Finally, to improve the practical usefulness of our approach, we plan to integrate our proposed solution into an IDE (e.g., Eclipse) and to evaluate the resulting tool by means of a user study. We also want to investigate the effectiveness of our approach under different experimental settings, e.g., evaluating on projects with a limited number of feature requests, cross-validating across projects, etc.

To extend our approach, we would like to consider how to enable users to specify some constraints to be taken into consideration in the recommendation process. We will also consider whether our proposed approach can be effective for bug reports in addition to feature requests.

Chapter 6

Recommending Code Changes for Automatic Backporting of Linux Device Drivers

6.1 Introduction

The Linux kernel today runs servers, desktop PCs, and laptops, as well as being at the heart of the Android OS, which runs the majority of smartphones, tablets, and a multitude of other devices. Device manufacturers increasingly find it important to have support for their products, in the form of *device drivers*, in the Linux kernel. The Linux kernel, however, is fast evolving, with frequent kernel-level API changes. This raises a challenge for device driver developers who have to choose a target kernel version that will be acceptable to the potential users of the device. A solution that helps ensure the continuing availability of the driver code is to target the current mainline version of the Linux kernel, so that the driver code can be integrated into the Linux kernel distribution itself and maintained by the mainline kernel developers [28]. Users, however, typically run older versions of the kernel, which are considered to be more stable. For such users, the driver must then be *backported* to older kernel versions.

Currently backports are typically done manually, on a case by case basis. An alternative is provided by the Linux backports project, which provides a compatibility library to hide differences between the current mainline and a host of older versions, and provides patches that allow a set of 800 drivers to target this compatibility library. These patches are either created manually by the backports project maintainers, or are created using manually written rewrite rules, based on the transformation tool Coccinelle [65]. In either case, however, the backports project maintainer has to determine where changes are needed in the code to backport and how to carry out these changes. Both of these operations are tedious and error prone.

While the Linux backports project provides partial automation, the user is limited to the versions for which backports have been prepared. In this chapter, we propose a step towards truly automating this task, in the form of a recommendation system for backporting driver files over code changes. Our approach accepts as input a driver file in a given Linux version, the older Linux version to which the driver file needs to be backported (the *target version*), and the git repository that stores the changes to the Linux source code. It first bisects the repository to find two subsequent commits in the repository such that compiling the driver file results in a compilation error in the older commit version and a successful compilation in the newer commit version. Next, our approach analyzes the differences between the two commits and compares them with the line of code containing the error, as indicated by the compiler. Our approach currently only considers cases where there is only one error line and analyzing these differences is enough to fix the error line and backport the driver file. Based on this analysis, our approach constructs a recommendation list that contains possible changes that can be applied to the error line to make the driver compilable in the target version. The changes are ranked by the similarity between the error line and the result of applying the change to the error line. Our experiment shows that, if a correct change exists in the recommendation list, it is often ranked highly.

The contributions of this work are as follows.

1. To the best of our knowledge, we are the first to work on recommending changes with the goal of automating backporting.

2. We propose a recommendation system that identifies a change in the code history that breaks the driver compilation, and recommends code change candidates that enable backporting of the driver code.

3. We evaluate our approach on 100 Linux device driver files. The recommendation list contains the correct backported code for 68 of the device driver files. Among these driver files, 73.5% of the correct recommendations are located in the Top-1 of the recommendation list.

The remainder of this chapter is structured as follows. We provide some background in Section 6.2. In Section 6.3, we present our proposed approach. We then describe our experiments in Section 6.4. Finally, we conclude in Section 6.5.

6.2 Preliminaries

In this section, we provide some background about the git version control system [22], and about GumTree [17], a tree differencing tool that we use in our approach.

6.2.1 Git

Git is a decentralized version control system that has recently become very popular due to the services that it provides and the tools that have been developed around it. Git is currently used by many software projects, including the Linux kernel. Git is designed around a workflow in which developers pull changes from other developers, modify their copy of the code on top of these changes, and request that other developers pull the changes that they have made. Pulling from another repository creates a *merge* node, representing the result of merging the two sets of changes. Technically, the commits are organized as a directed acyclic graph (DAG), although they are often collectively referred to as a *tree*.

To be sure to have access to complete information about earlier changes in a software project, we focus on the case where there is a single main developer of the system, with whom other developers want to regularly synchronize. The development of the mainline Linux kernel follows this model, as developers request that Linus Torvalds pull their changes prior to each release. In this case, we can view the commits and merges made by the main developer as a single primary trunk (level 1), and the commits made by other developers subsequent to pulling from the main developer and prior to requesting a pull from the main developer as being a secondary trunk (level 2), extending from the developer's initial pull to the merge. Such developers may furthermore serve as the main developer with respect to other developers, perhaps their local colleagues, leading to tertiary (level 3), quaternary (level 4), quinary (level 5), senary (level 6) trunks, etc. Figure 6.1 presents an example git tree illustrating pulls, merges, and primary, secondary, and tertiary trunks.

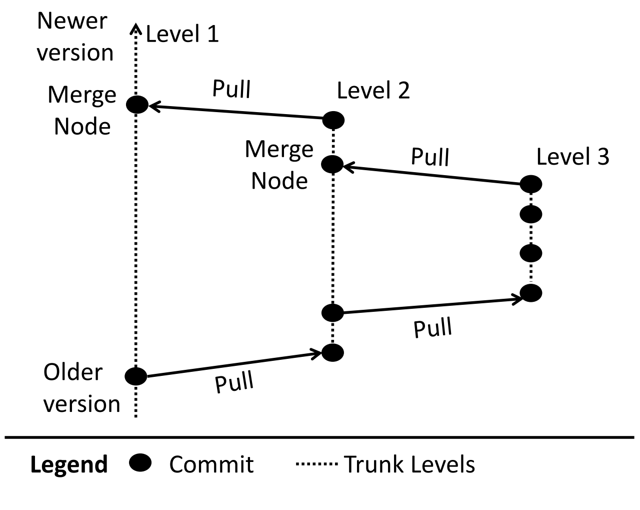


Figure 6.1: Example of a Git Tree

6.2.2 GumTree

A critical point of our approach is to be able to precisely identify changes between two commits in a code base. Line-based tools such as GNU `diff` are not sufficient, because a change can be mixed with irrelevant code fragments that happen to appear on the same line. To address this issue, we use tree differencing, as implemented by the tool GumTree [17]. GumTree identifies common subtrees in an abstract syntax tree, and then integrates common ancestors as long as there are not too many differences among their other descendants. A user study has found that the results of GumTree are considered to be better than those of a text-base differencing tool about half of the time, and mostly the same otherwise.

6.3 Proposed Approach

We first present a high-level overview of our automatic recommendation-based backporting approach and then elaborate on the different phases.

6.3.1 Overall Framework

As shown in Figure 6.2, our approach is divided into three phases: 1) error inducing change (EIC) search, 2) code transformation extraction, and 3) recommendation ranking. We define an error-inducing change as a patch between two consecutive commits in which compiling a target backport file in the older commit version leads to a compile error.

In the first phase, our approach gives as input a driver file that needs to be backported (*input driver file*), the Linux version to which we want to backport the driver file (*target Linux version*), and the git repository containing the change history between the target Linux version and the Linux version where the input driver file currently exists (*version control system*). The EIC search phase searches for two consecutive commits such that compiling the input driver file results in a compila-

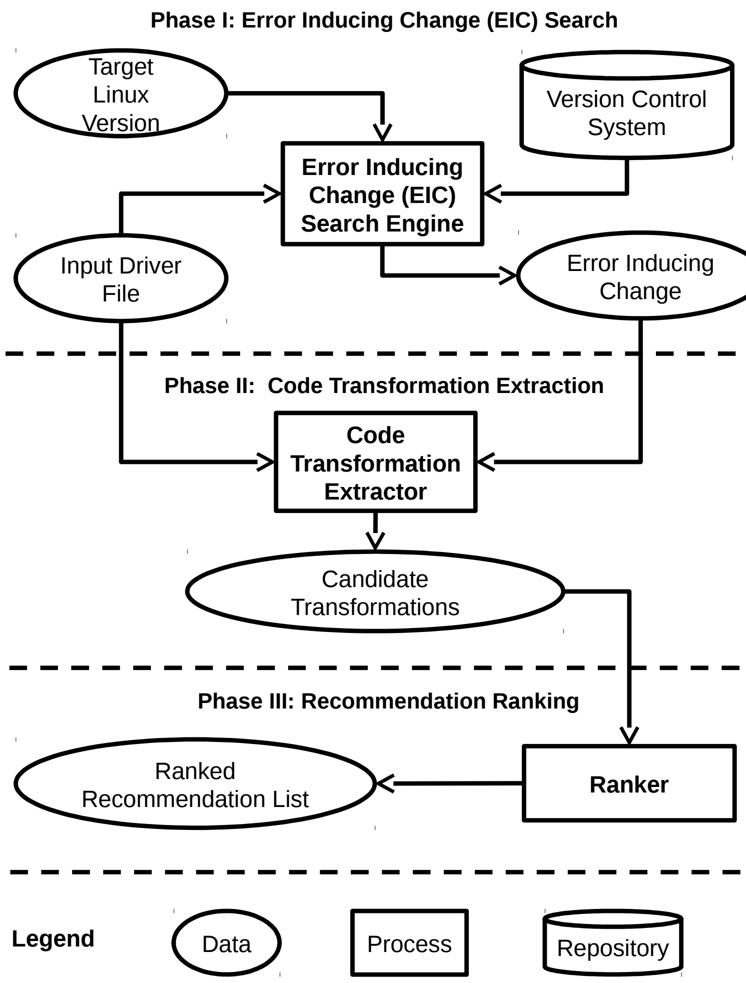

Phase I: Error Inducing Change (EIC) Search
Target Linux Version
Version Control System
Error Inducing Change (EIC) Search Engine
Input Driver File
Error Inducing Change
Phase II: Code Transformation Extraction
Code Transformation Extractor
Candidate Transformations
Phase III: Recommendation Ranking
Ranked Recommendation List
Ranker
Legend
Data
Process
Repository

Figure 6.2: Our Automatic Backporting Framework

tion error in the older commit version, and no error in the newer commit version. The goals of this phase are then two-fold: (1) Find the relevant change in the Linux kernel implementation that results in the input driver file not compiling in the target Linux version, (2) Find the changes that have been performed to existing Linux driver files to adapt them to this Linux kernel change. These adaptations are often committed at the same time as the relevant change to the underlying Linux kernel to prevent compilation errors. By reversing these adaptations, we can obtain hints on how to backport the input driver file.

The EIC search phase has one processing component, namely the *EIC Search*

Engine, which uses a binary search, starting with the target version, to jump through the change history recorded in the version control system and tries to compile the input driver file in each visited commit version. The search stops when it finds the commit version in which compiling the input driver file leads to no compilation error and the previous commit version still has compilation error. The change between this commit version and the previous commit version is the EIC. For example, an EIC may include the removal of a function definition on which the driver relies. Note that, in a more general case, there would be many EICs, however, we consider only cases where there is only one EIC. This one EIC may consist of many code changes between two consecutive commits that together render the code uncompilable. This EIC is output to the next phase.

In the second phase, our approach takes as input the EIC obtained in the previous phase and the input driver file. This phase searches for changes in the EIC that are relevant to the line in the input driver file that the compiler has marked as erroneous (which we refer to as the *error line*), and generates candidate transformations to backport the input driver file. This phase has one processing component, namely the *Code Transformation Extractor*, which matches the error line with each deleted line in the EIC. It then generates candidate transformations based on how the deleted lines are changed to the corresponding added lines. These candidate transformations are the recommendations produced by our approach.

In the third and final phase, our approach passes the candidate transformations to the *Ranker*, which ranks the transformations based on the similarity between the error line and the result of applying the transformation. We favor the minimal change between them. A developer who needs to backport the driver file can then examine the generated *ranked recommendation list* from top to bottom to find a suitable transformation.

We now describe each phase of our approach in more detail.

6.3.2 Error Inducing Change Search

To find the EIC, an approach could be to linearize the git commit history and perform a binary search on it. However, linearizing the history in any way would break the parent-child relationship between two consecutive commits, which represent the actual change made when a developer commits to its local repository. For example, linearizing commit history by date would result in a series of commits that is ordered by date. However, two consecutive commits in this case may not represent the actual change since the two commits might be made by two developers in their own local repository and just accidentally happen at around the same time.

Instead, we follow the approach presented in Algorithm 1. This algorithm takes as input the driver file DF to backport, the original Linux version $origRev$ for which the driver file has been implemented and the target Linux version $targetRev$ to which we want to backport the driver file.

The algorithm starts by retrieving the list $revList$ of all the commit hashes that lie in the main trunk ranging from the original version to the target version (Line 2). To achieve this, we use the command: git log –pretty=%H –first-parent ⟨target version⟩..⟨original version⟩. The option âĂŞpretty=%H causes the result to be a list of commit hashes and the option –first-parent causes the log to follow only the first parent commit after a merge. Next, at Line 3, the algorithm passes this list to the function EIC_Search, defined just below. EIC_Search first performs a binary search of the commit hashes in $revList$ to find a pair of consecutive commits, $child$ and $errParent$, such that the driver file does compile in the code resulting from $child$ but does not compile in the code resulting from $errParent$. Next, if $child$ is a merge commit (Line 6), then it must have some other parent in which the driver file does compile, because a merge commit does not itself change any code. In Line 7, EIC_Search checks each parent commit other than $errParent$ until it finds one in which the driver file compiles, which is named $okParent$. Based on our assumption that there is a single main developer from whom all code

Algorithm 1: Error Inducing Change Search

```
Input  : $DF$ = driver file to be backported
         $origRev$= the Linux version where $DF$ works fine
         $targetRev$ = target Linux version
Output: error inducing change
1  Main Algorithm Main_Search ($origRev, targetRev, DF$)
2      $revList = [origRev, targetRev]^{FirstParent}$
3      return EIC_Search ($revList$);
4  Procedure EIC_Search ($revList$)
5      $(child, errParent)$ = $Binary\_Search(revList)$
6      if child is a merge then
7          $okParent$=child's parent that can compile
8          $ancestor = findCommonAncestor(okParent, errParent)$
9          $newRevList = [okParent, ancestor]^{FirstParent}$
10         return $EIC\_Search(newRevList)$
11     else
12         return patch from child to the commit before it
13     end
```

is initially obtained, $errParent$ and $okParent$ have a common ancestor, which is named $ancestor$ in Line 8. This ancestor is obtained using the command: git merge-base $errParent$ $okParent$. EIC_Search then obtains the sequence of commit hashes from $ancestor$ to $okParent$. To achieve this, we use the command: git log –pretty=%H –first-parent $\langle ancestor\rangle..\langle okParent\rangle$. On the other hand, if the binary search in Line 5 yields a node that commits a code change, rather than a merge node, EIC_Search returns immediately with the patch from the commit node to its previous commit as the result (line 12).

Alternatively to Algorithm 1, we could potentially have used git bisect [23]. In our preliminary experiments, however, we have found that the commit returned by git bisect does not always have the property that the driver file to backport compiles in the returned commit and does not compile in its immediate predecessor. We will study this issue further in the future, but have chosen to rely on our algorithm, which integrates and ensures the property that we require.

Example: An example of the behavior of Algorithm 1 is illustrated in Figure 6.3. In this example, we take as an input a driver that works fine in the *Original* version. Our goal is to find the error inducing change that helps us backport the driver to the *Target* version, for which the driver currently cannot compile.

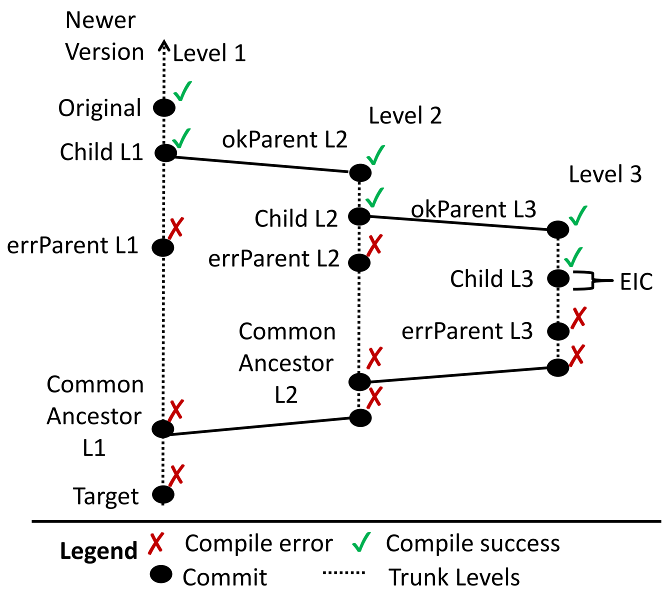


Figure 6.3: Example of our Error Inducing Change Search Algorithm

We start the search at the level 1 trunk, within the range of the *Original* version and the *Target* version. At this level, we perform binary search to find a pair of consecutive commits *Child L1* and *errParent L1* such that the driver file successfully compiles in the code resulting from *Child L1* but does not compile in the code resulting from *errParent L1*. In this example, since *Child L1* is a merge commit, it must have another parent in a different trunk level, for which the driver file successfully compiles. We denote this parent of *Child L1* as *okParent L2*. Next, we find *Common Ancestor L1*, which is the common ancestor of the *errParent L1* and *okParent L2*. Afterwards, we perform another binary search on the range of commits between *okParent L2* and *Common Ancestor L1* to find a pair of consecutive commits *Child L2* and *errParent L2*, such that the driver compiles with the code resulting from *Child L2* but does not compile with the code resulting from *errParent L2*. We repeat the search in a similar manner until we find a pair of consecutive commits *Child L3* and *errParent L3*, such that *Child L3* is not a merge commit and the driver file compiles successfully in the code resulting from *Child L3*, but does not compile in the code resulting from *errParent L3*. Finally, we report *Child L3* as the error inducing change.

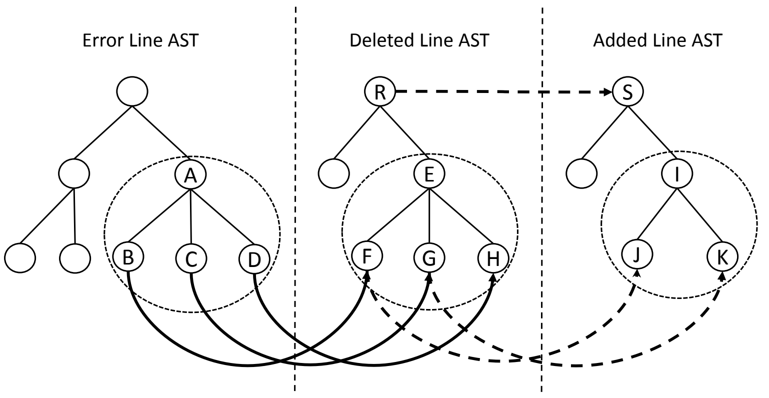


Figure 6.4: Code transformation extraction illustration. Error Line AST is extracted from the driver file. Deleted Line and Added Line AST are extracted from historical changes found in the Linux kernel.

6.3.3 Code Transformation Extraction

The *Code Transformation Extractor* processes the input driver file and the error inducing change (EIC) obtained by the previous phase. First, we compile the driver file in the target Linux version. By the definition of the backporting problem, we know that this will result in a compilation error. We then record the contents of the line containing the error. The goal is to find a change in the EIC that can be applied to the error line to remove the compiler error. The EIC can be viewed as a code difference (*diff*) composed of *hunks*, each of which is a contiguous sequence of lines corresponding to a sequence of line deletions, line additions, or both. Since we consider a backporting setting (changing code that works on a newer version to adapt it to an older version), to view a transformation in a natural way, we reverse the direction of a normal patch and thus a deleted line is a line in the newer code whereas an added line is a line in the older code. A hunk containing only line additions or deletions does not correspond to a code change from which we can infer a transformation. Thus, we exclude such hunks from our analysis. We also ignore hunks that appear in non source-code files.

The *Code Transformation Extractor* constructs an Abstract Syntax Tree (AST) for the error line and for each deleted line and each corresponding added line in the EIC. In general, a line in the source file might not represent a complete term in

Algorithm 2: Code Transformation Extraction

```
Input  : $AST_{Err}$ = AST of the error line
         $AST_{Del}$ = AST of a deleted line
         $AST_{Add}$ = AST of an added line
Output: A code transformation or $\varnothing$
1  $(N_E, M_D)$ = Tree differencing ($AST_{Err}$, $AST_{Del}$);
2  $(N_D, M_A)$ = Tree differencing ($AST_{Del}$, $AST_{Add}$);
3  if $N_E \neq \{\}$, $M_D \neq \{\}$, $N_D \neq \{\}$, $M_A \neq \{\}$, and $M_D \cap N_D \neq \{\}$ then
4      $ST_{Err}$ = Smallest subtree in $AST_{Err}$ covering all mapped nodes;
5      $ST_{Del}$ = Corresponding subtree of $ST_{Err}$ in $AST_{Del}$;
6      $ST_{Add}$ = Corresponding subtree of $ST_{Del}$ in $AST_{Add}$;
7      $varMap$ = Matched identifier mappings from $AST^{Err}$, $AST^{Del}$, $AST^{Add}$;
8      $ST^{M}_{Add}$ = Apply $varMap$ to $ST^{Add}$;
9      Output ($ST_{Err} \Rightarrow ST^{M}_{Add}$);
10 else
11     Output $\varnothing$;
12 end
```

the C language. Since the complete source files are available, via git, our approach constructs an AST for the function containing each line, and then extracts from that the smallest sub-AST that contains all of the tokens of the considered line. The Code Transformation Extractor then tries to map the nodes of the AST of the error line to nodes of the AST of every deleted line, and the nodes of the AST of a deleted line to the nodes of the AST of every added line in the same hunk. For each combination of error line, deleted line, and added line, the *Code Transformation Extractor* identifies a subtree in the error line's AST that matches a subtree in the deleted line's AST, which in turn matches a subtree in the added line's AST. For each matching, we transplant the matching subtree in the added line's AST to the error line's AST and adapt it with any necessary substitutions of identifier names. The adapted transplantation amounts to a transformation. A set of such possible transformations is output to the next phase.

For each combination of error line, deleted line, and added line, *Code Transformation Extractor* follows Algorithm 12 to produce zero or one candidate transformations. This process involves three main steps: node mapping, extraction of matching subtrees, and subtree transplantation and subterm replacement.

Node Mapping Given an error line's AST, a deleted line's AST, and an added line's AST, we want to find two node mappings: (1) between the nodes in the error line's AST and the deleted line's AST, and (2) between the deleted line's AST and the added line's AST. For this, we use the tree differencing tool GumTree [17] (lines 1-2), described in Section 6.2.2. GumTree maps nodes in the two input ASTs based on some heuristics. It outputs a set of mapped nodes, denoted as $(N, M) = \{(N_1, M_1), ..., (N_k, M_k)\}$ where $N = \{N_1, ..., N_k\}$ and $M = \{M_1, ..., M_k\}$ are the mapped nodes in the two ASTs and k is the number of mapped nodes. If the set of mapped nodes between the error line's AST and deleted line's AST ((N_E, M_D)) and the set of mapped nodes between the deleted line's AST and the added line's AST ((N_D, M_A)) are both non-empty, and the intersection of the mapped nodes in the deleted line's AST based on the two differencing operations is non-empty ($M_D \cap N_D \neq \{\}$), then we proceed with the next steps (Lines 3-9), otherwise we output no transformation (Line 11).

Extraction of Matching Subtrees Given a mapping of nodes in the ASTs, our approach next extracts the minimal subtrees that cover all the mapped nodes. Our approach first identifies the minimal subtree in the error line's AST (Line 4) and the corresponding subtree in the deleted line's AST (Line 5). Next, it finds nodes in the added line's AST that are mapped to the minimal subtree in the deleted line's AST and then extracts a minimal subtree that covers these mapped nodes (Line 6).

As an example, we can employ GumTree to generate the mapping between the error line's AST and deleted line's AST in Figure 6.4 (solid arrows in the figure). There are three nodes that are mapped between these ASTs: B to F, C to G, and D to H. We then extract the smallest subtree of each tree that covers all of the mapped nodes. This gives the ABCD subtree for the error line's AST and the EFGH subtree for the deleted line's AST.

Next, we again can employ GumTree to generate the mapping between EFGH subtree in the deleted line's AST and the added line's AST in Figure 6.4 (dashed arrows in the figure). There are three mapped nodes: R is mapped to S, F is mapped to J, and G is mapped to K. Our approach only focuses on the EFGH subtree since it is the only part that matches with the error line's AST and thus, for fixing the error code, our approach only needs to care about the mappings inside this subtree. Next, our approach marks the relevant mapped nodes in the added line's AST and extracts a subtree, IJK, that covers all of the marked nodes. ABCD, EFGH, and IJK are the matching subtrees that are output to the next step.

Subtree Transplantation and Subterm Replacement In this step, we have three matching subtrees, ST_{Err}, ST_{Del}, and ST^{Add}, extracted from the error line's AST, the deleted line's AST, and the added line's AST, respectively. Our approach generates a candidate transformation by transplanting an adapted ST^{Add} to replace ST^{Err} in the error line's AST. Adaptation to ST^{Add} is needed since subterms used in ST^{Add} may differ from those used in ST^{Err}. We infer the necessary replacements of subterms from the mapped nodes between ST^{Err} and ST^{Del}, and between ST^{Del} and ST^{Add}. For a term v in node n in ST^{Err} that is mapped to n' in ST^{Del} (that contains term v'), which is subsequently mapped to n'' in ST^{Add} that contains term v', our approach will store a replacement (v,v') (Line 7). Each replacement (v,v') is applied to nodes in ST_{Add} to create ST^{M}_{Add} where any subterm v' is replaced with v (Line 8). We will then output a transformation ($ST_{Err} \Rightarrow ST^{M}_{Add}$) which corresponds to the transplantation of the adapted subtree (Line 9). At this point, our approach has essentially learned a transformation from an example.

To illustrate how Algorithm 12 works, consider Figure 6.4. In that figure, we need to transplant the adapted IJK subtree to replace the ABCD subtree in the error line's AST. The IJK subtree is adapted by renaming subterms in the subtree with their corresponding mapped subterms. The subterms are inferred from the mapped nodes between ABCD and EFGH subtrees and EFGH and IJK subtrees. Consider any subterms a and b in the ABCD subtree, any subterms c and d in the EFGH subtree, and any subterm c in the IJK subtree. The mapping may then indicate that

a in the ABCD subtree is mapped to c in the EFGH subtree which in turn is mapped to c in IJK subtree. Our approach then creates a replacement (a,c) and will replace all subterms c in IJK with a before transplanting IJK to the error line's AST. The corresponding transformation is (ABCD $\Rightarrow$ IJK[$c$$\rightarrow$$a$]).

The above illustration applies to many kinds of changes, such as changes in the field/constant accessed, an argument of a function, the name of a function, or the condition of an if. As a concrete example, we consider a function name change:

Error Line:

```
node=acpi_ns_validate_handle(target_handle);
```

Deleted and Added Lines:

```
- node=acpi_ns_validate_handle(obj_handle);âĂŃ
+ node=acpi_ns_map_handle_to_node(obj_handle);âĂŃ
```

Matching the error line AST against the deleted line AST matches `node` with `node`, = with =, `acpi_ns_validate_handle` with `acpi_ns_map_handle_to_node`, `(` with `(`, `target_handle` with `obj_handle`, and `)` with `)`. All of the tokens are accounted for. In this special case, the subtree transplantation will basically replace the error line with the added line. However, the subterm `obj_handle` does not exist in the code containing the error line. Thus, we perform a replacement, (`obj_handle`, `target_handle`), which is learned from the mapping. At this point, we have transplanted the added line's subtree to the error line AST and adapted the corresponding subterm to match the one found in the error line.

6.3.4 Recommendation Ranking

Ranker takes as input the set of candidate transformations produced by the previous phase and applies each one to the error line in the input driver file to produce a changed error line. Following Occam's razor, our intuition is that the correct transformation is likely to be the one that transforms the error line to something that remains similar to it.

To compute the similarity between the error line and a changed error line, *Ranker* takes the sequence of strings in the error line and changed error line and removes all whitespace from both sequences. It then compares the resulting sequences of characters using Ratcliff and Obershelp's string alignment algorithm [9], which finds a semi-optimal matching of characters in two sequences. Specifically, the algorithm first finds the longest contiguous subsequence between the two sequences. It then recursively processes the subsequences to the left and right of the longest contiguous subsequence. After the matching characters between the two sequences are found, the similarity between the two sequences $SeqSim$ is computed as follows:

$$SeqSim = \frac{2 \times M}{T}$$

where T is the total number of characters in both sequences, and M is the number of matched character pairs. As an example, consider two sequences "abcd" and "bcde". Both sequences have "bcd" sequence as their string subsequence. Thus, T=4 and M=3, and the value of $SeqSim$ is 0.75.

Ranker ranks the transformations by decreasing $SeqSim$ values, and then outputs a *ranked recommendation list*.

6.4 Experiments & Analysis

In this section, we first describe our dataset, evaluation measure, and experimental settings. We then list our research questions and provide our experimental results. We finish with a discussion and a description of the threats to validity.

6.4.1 Dataset

Our dataset consists of 100 device driver files from Linux 2.6.x versions. Using this dataset, we intend to simulate a backporting scenario. Since general automatic backporting is a new and hard problem, we limit the problem to make it more feasible to solve. Specifically, we select driver files and starting and target versions according to the following criteria.

1. The driver file should have only one changed line of source code between two consecutive Linux versions, e.g., Linux versions 2.6.1 and 2.6.2. We focus on one-line changes to limit the difficulty of making the new code work in the older version. We consider that one-line changes represents a reasonable difficulty for an initial attempt at automatic backporting.

2. Following the first criteria, the driver file will be present in two versions: the old and new versions. We should be able to compile the old and new versions of the driver file in their corresponding Linux kernel versions.

3. When we compile the new version of the driver file within the old version of the Linux kernel, a compilation error should occur. Our goal is to modify the the a copy of the new driver to fix this compilation error.

Table 6.1 shows the distribution of the 100 Linux driver files selected according to the above criteria.

6.4.2 Evaluation Measure

We use Hit@N to measure the effectiveness of our ranking strategy. We define it as follows:

$$Hit@N = \begin{cases} 1, & \text{if } CC \text{ is in the Top-N of } RecCC. \\ 0, & \text{otherwise.} \end{cases}$$

where CC denotes the correct code change, $RecCC$ denotes a code change recommendation list containing the correct code change, and N denotes the number

Table 6.1: Distribution of the 100 Driver Files in Our Dataset across Device Driver Families

Driver Family	**#Drivers**	**Driver Family**	**#Drivers**	**Driver Family**	**#Drivers**
ata	24	char	3	serial	1
media	15	md	3	s390	1
net	11	mtd	3	usb	1
gpu	7	spi	2	power	1
bluetooth	5	hid	2	cpuidle	1
isdn	5	leds	2	ide	1âĂŃ
infiniband	4	scsi	2		
acpi	4	xen	1		

of entries considered at the top of the recommendation list. We compute the average Hit@N across the recommendations to measure the effectiveness of our ranking strategy.

6.4.3 Experimental Settings

We simulate a backporting scenario for the 100 driver files in our dataset. For each driver file, we pretend that the driver file is new and has never existed before in the Linux kernel source code. We use the command `make defconfig` to prepare a configuration in which to compile the kernel. We then apply our approach and find the error inducing change for this input driver file. Since the change to the driver file itself is also included in the version control system history, we exclude this change if it appears in the error inducing change. We then find the candidate changes for the input driver file using the remaining examples. To check whether our backport is correct, we simply compare the backported code with the actual old version of the driver file. We consider that the backport is successful only if the backported code is exactly the same as the old version of the code.

6.4.4 Research Questions

Research Question 1. *How effective is our proposed approach in extracting correct code changes for a given device driver file?* We report the accuracy of our approach in extracting the correct change. We also investigate the distribution of extracted correct code changes among different driver families.

Research Question 2. *When the correct code change exists in the recommendation list, how high is it ranked by our approach?* We then investigate the rankings of correct code changes in this set. The higher the rankings, the better the recommendations. We measure the recommendation effectiveness using the average Hit@N metric.

Research Question 3. *In what kinds of cases can our approach extract the correct code changes?* We show some cases where our approach extracts correct code changes and assess why our approach works well in these cases.

Research Question 4. *In what kinds of cases is our approach unable to extract the correct code changes?* We show some cases where our approach extracts either incorrect changes or nothing at all. These cases can guide future work.

6.4.5 RQ1: Effectiveness of Code Change Extraction

Our approach produces correct code changes to successfully backport 68 out of 100 drivers, giving a success rate of 68%. Table 6.2 shows the distribution of the successful cases. Our approach successfully extracts all required code changes for all selected drivers from 7 driver families: ata, bluetooth, mtd, spi, scsi, serial, and cpuidle. For many other remaining driver families, it is successful on some and fails on others.

Table 6.2: Distribution of correct code changes that are successfully extracted per driver family. The number in parentheses is the total number of selected drivers in the driver family.

Driver Family	#Successful	Driver Family	#Successful
ata	24 (24)	scsi	2 (2)
media	10 (15)	serial	1 (1)
bluetooth	5 (5)	cpuidle	1 (1)
gpu	4 (7)	md	1 (3)
net	4 (11)	xen	0 (1)
infiniband	3 (4)	s390	0 (1)
isdn	3 (5)	usb	0 (1)
acpi	3 (4)	power	0 (1)
mtd	3 (3)	hid	0 (2)
char	2 (3)	leds	0 (2)
spi	2 (2)	ide	0 (1)

Table 6.3: Effectiveness of Our Ranking Approach

N	#Correct Code Changes	Average Hit@N
1	50	0.735
2	58	0.853
3	58	0.853
4	58	0.853
5	60	0.882

6.4.6 RQ2: Effectiveness of Code Change Ranking

Table 6.3 shows the effectiveness of our ranking strategy, for the 68 driver files for which our approach extracts correct code changes. By only recommending the Top-1 code change, we recommend the correct code change for 50 out of the 68 driver files, giving an average Hit@1 of 0.735. Increasing the recommendation to Top-2, we find that there are 8 more driver files whose correct code changes are recommended. This translates to an average Hit@2 of 0.853. Increasing the recommendation to Top-3 and Top-4 does not change the number of driver files for which the correct code change is recommended. When the recommendation is increased to Top-5, there are 2 more such driver files. Thus, by only recommending Top-5 candidate code changes, our approach can successfully find the correct code change 88.2% of the time, thus achieving an average Hit@5 of 0.882.

6.4.7 RQ3: Cases where Correct Code Changes are Successfully Extracted

Case 1: Change of record access.

Driver File: drivers/char/drm/drm_agpsupport.c

Error Line:

```
return drm_agp_acquire(
   (struct drm_device*) file_priv->minor->dev);
```

Change example:

```
- struct drm_device *dev = priv->minor->dev;
+ struct drm_device *dev = priv->head->dev;
```

Corrected Error Line:

```
return drm_agp_acquire(
   (struct drm_device*) file_priv->head->dev);
```

In this case, *file_priv→minor→dev* is the part of the error line that provokes a compile error. This code matches *priv→minor→dev* in the deleted line of the change example. This portion of the deleted line matches *priv→head→dev* in the added line. Thus, the *file_priv→minor→dev* portion of the error line is replaced with *priv→head→dev* from the added line. However, we know that the identifier *priv* in the deleted line matches the identifier *file_priv* in the error line. Based on the change example, we also know that *priv* is not changed. Thus, *file_priv* should not change either. We thus map back *priv* to *file_priv*, which is found in the corresponding context in the error line.

Case 2: Deletion of a function argument.

Driver File: drivers/ata/pata_artop.c

Error Line:

```
return ata_pci_sff_init_one(pdev,ppi,&artop_sht,NULL,0);
```

Change Example:

```
- return ata_pci_sff_init_one(dev,ppi,&generic_sht,NULL,0);
+ return ata_pci_sff_init_one(dev,ppi,&generic_sht,NULL);
```

Corrected Error Line:

```
return ata_pci_sff_init_one(pdev,ppi,&artop_sht,NULL);
```

When we match the error line with the deleted line in the change example, the entire *ata_pci_sff_init_one* function call in the former matches the *ata_pci_sff_init_one* function call in the latter. *pdev* of the error line matches *dev* of the deleted line. Similarly, *ppi* is mapped to *ppi*, *&artop_sht* to *&generic_sht*, *NULL* to *NULL*, and 0 to 0. The matched portion of the error line is then replaced with the matched portion of the added line. We then rename the variables in the matched portion of the added line by changing *dev* to *pdev* and *generic_sht* to *artop_sht*.

Case 3: Change of function name.

Driver File: drivers/acpi/acpica/nsnames.c

Error Line:

```
    node=acpi_ns_validate_handle(target_handle);
```

Change Example:

```
    - node=acpi_ns_validate_handle(obj_handle);
    + node=acpi_ns_map_handle_to_node(obj_handle);
```

Corrected Error Line:

```
    node=acpi_ns_map_handle_to_node(target_handle);
```

In the above example, the entire error line has the same structure as the entire deleted line. The function names from the two lines are matched and the argument *target_handle* is matched with the argument *obj_handle*. After the algorithm replaces the error line portion with the added line portion, the function name *acpi_ns_validate_handle* will be changed to *acpi_ns_map_handle_to_node*. To complete the backport, *obj_handle* is renamed to *target_handle*.

Case 4: Change of constant.

Driver File: drivers/media/video/cx23885/cx23885-input.c

Error Line:

```
rc_map=RC_MAP_HAUPPAUGE;
```

Change Example:

```
- dev->init_data.ir_codes = RC_MAP_HAUPPAUGE;
+ dev->init_data.ir_codes = RC_MAP_RC5_HAUPPAUGE_NEW;
```

Corrected Error Line:

```
rc_map=RC_MAP_RC5_HAUPPAUGE_NEW;âĂŃ
```

In this example, again the entire error line matches the entire deleted line, with *rc_map* matched with *dev→init_data.ir_codes*, the assignment node in the error line with the assignment node in the deleted line, and *RC_MAP_HAUPPAUGE* in the error line with the occurrence of the same constant in the deleted line. The error line is then replaced with the added line. To complete the backport, *dev→init_data.ir_codes* is renamed to *rc_map*.

Case 5: Change of if condition.

Driver File: drivers/ata/pata_oldpiix.c

Error Line:

```
if(ata_dma_enabled(adev))
```

Change Example:

```
- if(ata_dma_enabled(adev))
+ if(adev->dma_mode)âĂŃ
```

Corrected Error Line:

```
if(adev->dma_mode)âĂŃ
```

In this case, the if condition in the error line is identical to the one in the deleted line. All matched nodes are of the same type and name. Thus, we directly replace the if condition in the error line with the if condition in the added line to perform the backport.

6.4.8 RQ4: Cases where Correct Code Changes are Not Successfully Extracted

Case 1: The correct transformation needs to be learned from multiple deleted-added line pairs.

Driver File: drivers/net/wireless/ath/ath9k/virtual.c

Error Line:

```
txctl.frame_type =
    ps ? ATH9K_IFT_PAUSE : ATH9K_IFT_UNPAUSE;
```

Corrected Error Line:

```
txctl.frame_type =
    ps ? ATH9K_INT_PAUSE : ATH9K_INT_UNPAUSE;
```

In this case, the transformation of the error line into the corrected line involves two changes: transforming *ATH9K_IFT_PAUSE* into *ATH9K_INT_PAUSE*, and transforming *ATH9K_IFT_UNPAUSE* into *ATH9K_INT_UNPAUSE*. Each of these transformations can be obtained from different deleted-added line pairs in the EIC. Although both transformations exist in the EIC, our approach can currently only learn from a single deleted-added line pair. Thus, it applies only one of the two transformations and fixes the error line partially.

Case 2: The EIC provides no relevant example.

The EIC may only contain changes in the definitions that are used by the error line, but all other code that used these definitions could have been updated in earlier commits. Thus, we might not find similar lines of code in the EIC that would suggest how to fix the error line. For such cases, our approach is not be able to recommend correct code changes.

6.4.9 Discussion

The quality of our automatic backporting approach depends on the capabilities of each phase in our framework. We assess the possible limitations in each case.

The error inducing change search phase is intended to find a change that contains an example from which we can learn how to backport the code. However, as noted above, all such examples may occur in earlier commits. To address this issue, we would need to extend our search to relevant commits that occurred prior to the error inducing change.

The error transformation extraction phase is intended to learn a potential candidate transformation from a change between a pair of added and deleted lines. Our evaluation shows that there are some cases in which learning from only one pair of added and deleted lines is not enough. Thus, we need a capability to decide which transformation can be combined with another transformation to make a recommendation.

Our approach ranks the candidate transformations, in the ranking recommendation phase, because we have no way to know for sure which transformation is the correct one. A possible way to check the transformation's correctness would be through the use of test cases. Nevertheless, device drivers are difficult to test automatically, because doing so requires installing the device driver on an OS connected to a real hardware or an emulator, and checking whether the hardware is detected and that all of its functionalities can be accessed by the OS. And even if test cases existed for the new driver, they might not be directly usable in the older version. Thus, we would need to to backport the test cases and check whether their backport is correct, repeating the same problem.

Regarding the transformation results, although some may look simple, it is not trivial for developers to perform them. Many people have contributed to the Linux kernel, and around 70,000 commits were made on the Linux kernel in 2015 alone. A developer is unlikely to be knowledgable about the large number of diverse changes made by others. Moreover, state-of-the-art works in automated bug fixing also can only fix a small number of bugs that span one or a few lines of code [31, 42], showing the level of difficulty of automatically inferring correct fix for even one line of code. Although bug fixing and backporting are two different problems, both involve transforming a broken piece of code to another that works.

Last but not least, our approach currently targets backports that require only a one-line change. We have evaluated it on a simulation where we backport a driver file across two consecutive Linux versions. Analyzing a *diff* of two consecutive Linux versions, for all versions between 2.6.11 to 3.13.3, we found 944 driver files having a one line change that affects the driver functionality, meaning that using the newer version of the driver in the previous version of Linux will not work. Thus, even though one-line change is very limited, it still covers quite a number of potential backporting situations.

Another limitation in our evaluation is that we assume that syntactic correctness equals semantic correctness. Since our simulation is based on actual changes, we believe this assumption holds as we only consider backporting to be successful if the resulting code is exactly the same.

In a more general case, a developer may be called upon to backport a driver that requires more than one line of changes. Indeed, the changes required may involve not only multiple lines, but also multiple files, and may require a deeper understanding of the relationships between them. Our approach could still be helpful if the task can be broken down into individual issues that involve only one error line.

6.4.10 Threats to Validity

Possible threats to validity include threats to construct validity, to internal validity, and to external validity.

Threats to Construct Validity. These threats refer to the suitability of our evaluation measures. We make use of accuracy and Hit@N as the effectiveness measures of our approach. These measures have been used in past studies and thus we believe the threats are minimal.

Threats to Internal Validity. These threats correspond to experiment errors. To reduce the likelihood of this threat, we have checked the implementation of our approach multiple times. Still, there may be errors that we missed.

Threats to External Validity. These threats refer to the generalizability of our experimental results. We have only investigated 100 Linux device driver files and backporting cases that involve two successive Linux versions. The effectiveness of our approach beyond these driver files and for two arbitrary Linux versions is not guaranteed to be the same. In the future, we plan to reduce this threat by evaluating our approach on more device driver files and arbitrary pairs of Linux versions.

6.5 Conclusion

In this chapter, we have presented a recommendation system that generates a ranked list of candidate transformations for (semi-)automatic backporting of Linux device drivers. Our approach consists of three phases. First, we search for a change that can give us a clue on how to fix the error when we backport a driver to an older Linux version. Then, we extract code transformation candidates that we may apply to fix the error. Finally, we rank the transformation candidates. Our simulated backporting experiment on 100 Linux device drivers shows that our approach can extract the correct transformation for 68% of the device drivers. Among the device drivers having a correct transformation in the recommendation list, our ranking approach ranks first 73.5% of the correct transformations. We then illustrate some successes and limitations of our approach.

In future work, we plan to improve our search algorithm by extending our search beyond the error inducing change, especially to older changes involving the modified data structure or function in the error inducing change. We plan to extend our code transformation extraction algorithm to be able to learn from many deleted-added line pairs. We also plan to consider a more general backporting scenario, such as multi line and multi file backporting. We also plan to explore possibility of lightweight testing for improving the correctness of the backported device drivers. Finally, we plan to experiment on a bigger dataset to ensure the generalizability of our approach.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the contributions of our works. We then describe the future work in recommending APIs for software evolution.

7.1 Summary of Contributions

Finding and using relevant APIs for software evolution are not always easy. The process can be time consuming as the search space for finding APIs is often huge and understanding relevant APIs may require some learning time and effort. To help developers easily use APIs, we have developed four approaches that tackle different problems in API recommendation:

1. Our first work recommends API libraries given a known to be useful or existing set of libraries in the system [77]. This recommendation system can help developers to find additional relevant APIs. The recommendation system is built by combining association rule mining and collaborative filtering techniques. Experiments with the approach show that it can achieve a recall rate@5 of 0.852 and a recall rate@10 of 0.894 for recommending additional libraries.

2. Our second work is based upon our first work. We propose a new library recommendation approach called *LibXplore*. This approach combines the power

of collaborative filtering, association rule mining, and matrix factorization. It considers a set of existing libraries as input, and historical library usage and textual content from initial libraries as knowledge source. We evaluate the approach by mining the actual set of initial and the historically added libraries from version control repository. Our approach improves upon our previous approach by 20.44% and 9.69%, in terms of Hit@5 and Hit@10, respectively.

3. Our third work recommends API methods given a target library and a textual description of a task [76]. This recommendation system can help developers to find relevant API methods in the given library that can be used to realize the task. The recommendation system is built by combining information retrieval and collaborative filtering techniques. Experiment with the approach shows that it can achieve a recall-rate@5 of 0.690 and a recall-rate@10 of 0.779 for recommending API methods.

4. Our fourth work recommends APIs in a different way [78]. Rather than handling the usual problem of implementing new code for a software, this work focuses on making the features or bug fixes implemented in the new code available in an older version of the software. Our approach can provide recommendations on how to transform the new code to its equivalent form in the old version. One important part of the problem is to *adapt API invocations* in the new code so that it can work with APIs available in the older version of the software. Experiments with the approach show that it can extract correct transformations for 68% of the tested device drivers and put 73.5% of correct transformations in the first rank.

7.2 Future Work

There are many potential future works that can be pursued for API recommendation. We present some of them below:

1. Our API library recommendation system assumes that some API libraries are already used in the software or the software developers have some ideas of the API libraries that they can use for building the software. We plan to develop another version of API library recommendation that accepts a software description as input and recommends relevant libraries according to the software description.

2. We want to improve our API method recommendation by deeply interpreting the meaning of the natural language description of the software. This description is often in a very high level semantics while APIs are usually in a lower level semantics. For example, a software description might talk about file management. We want to bring the description to lower level semantics such as creating file, deleting file, etc.

3. The next natural step after recommending API methods is recommending API method parameters. Given an API method, we plan to recommend relevant parameters by learning the specification of the required parameters and generate concrete parameters by looking at historical data and existing context of the software code (if available).

4. Given a list of API methods for a certain task, we plan to glue the methods together to achieve the task. In creating the glue code, we plan to use program synthesis, especially for variable creations and conversions of one type of variable to another. We would be also using an API method parameter recommendation approach inside this approach.

5. For our backporting work, we plan to improve our recommendation by mining change rules and use them together with our current approach.

Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Int. Conf. on Very Large Data Bases*, 1994.

[2] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 286–295. ACM, 2016.

[3] Jesper Andersen and Julia L. Lawall. Generic patch inference. *Autom. Softw. Eng.*, 17(2):119–148, 2010.

[4] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 382–385, 2012.

[5] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[6] Yves Bastide, Nicolas Pasquier, Rafik Taouil, Gerd Stumme, and Lotfi Lakhal. Mining minimal non-redundant association rules using frequent closed itemsets. In *International Conference on Computational Logic*. 2000.

[7] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. A structured approach to assess third-party library usage. In *ICSM*, pages 483–492, 2012.

[8] Tegawendé F. Bissyandé, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, and Laurent Réveillère. Empirical evaluation of bug linking. In *CSMR*, pages 89–98, 2013.

[9] Paul E Black. Ratcliff/Obershelp pattern recognition. *Dictionary of Algorithms and Data Structures*, 2004.

[10] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3, 2003.

[11] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *ESEC/SIGSOFT FSE*, pages 213–222, 2009.

[12] George Casella and Edward I. George. Explaining the Gibbs sampler. *The American Statistician*, (3):167–174, 1992.

[13] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching connected api subgraph via text phrases. In *SIGSOFT FSE*, 2012.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[15] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. Automatically generating test cases for specification mining. *IEEE Trans. Software Eng.*, 38(2), 2012.

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

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, (ASE)*, pages 313–324, 2014.

[18] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, 2009.

[19] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering (TSE)*, 33(11):725–743, 2007.

[20] Harald C Gall, Beat Fluri, and Martin Pinzger. Change analysis with :Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, 2009.

[21] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in IR-based concept location. In *ICSM*, pages 351–360, 2009.

[22] Git. `http://git-scm.com/`.

[23] Git bisect. `http://git-scm.com/docs/git-bisect`.

[24] Negar Hariri, Carlos Castro-Herrera, Jane Cleland-Huang, and Bamshad Mobasher. Recommendation systems in requirements discovery. In *Recommendation Systems in Software Engineering*, pages 455–476. Springer, 2014.

[25] Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *ASE*, pages 524–527, 2011.

[26] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1990.

[27] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *DSN*, pages 52–61, 2008.

[28] Greg Kroah-Hartman. The Linux kernel driver interface (all of your questions answered and then some). `https://www.kernel.org/doc/Documentation/stable_api_nonsense.txt`.

[29] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Mining message sequence graphs. In *ICSE*, 2011.

[30] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *SAC*, pages 1317–1324, 2011.

[31] Xuan-Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of the 23nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016.

[32] Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

[33] Jinyan Li, Haiquan Li, Limsoon Wong, Jian Pei, and Guozhu Dong. Minimum description length principle: Generators are preferable to closed patterns. In *AAAI*, pages 409–414, 2006.

[34] Wayne C Lim. Effects of reuse on quality, productivity, and economics. *IEEE software*, 11(5):23–30, 1994.

[35] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[36] David Lo and Siau-Cheng Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, pages 265–275, 2006.

[37] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining past-time temporal rules from execution traces. In *WODA*, pages 50–56, 2008.

[38] David Lo and Shahar Maoz. Specification mining of symbolic scenario-based models. In *PASTE*, pages 29–35, 2008.

[39] David Lo and Shahar Maoz. Mining hierarchical scenario-based specifications. In *ASE*, 2009.

[40] David Lo and Shahar Maoz. Scenario-based and value-based specification mining: better together. In *ASE*, 2010.

[41] David Lo, Ganesan Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. In *WCRE*, pages 62–71, 2009.

[42] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 298–312. ACM, 2016.

[43] Fan Long, Xi Wang, and Yang Cai. Api hyperlinking via structural overlap. In *ESEC/SIGSOFT FSE*, pages 203–212, 2009.

[44] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[45] C.D. Manning, P. Raghavan, and H. Schutze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008.

[46] Paul Marinescu, Petr Hosek, and Cristian Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 93–104. ACM, 2014.

[47] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *ICSE*, pages 364–374, 2012.

[48] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[49] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *34th International Conference on Software Engineering (ICSE)*, pages 353–363. IEEE, 2012.

[50] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12(5):471–516, 2007.

[51] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *International Workshop on Mining Software Repositories (MSR)*, pages 1–5. ACM, 2005.

[52] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 803–813. ACM, 2014.

[53] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *ASE*, pages 70–79, 2012.

[54] Johan Natt och Dag, Vincenzo Gervasi, Sjaak Brinkkemper, and Bjorn Regnell. Speeding up requirements management in a product software company:

Linking customer wishes to product requirements through linguistic engineering. In *RE*, 2004.

[55] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In Catriel Beeri and Peter Buneman, editors, *ICDT*, 1999.

[56] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. Codetube: extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 645–648. ACM, 2016.

[57] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[58] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, and G. Antoniol. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. In *TSE*, 2007.

[59] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K Roy. Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 194–203. IEEE, 2014.

[60] Shivani Rao and Avinash C. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.

[61] Romain Robbes and Michele Lanza. Improving code completion with program history. *Autom. Softw. Eng.*, 17(2):181–212, 2010.

[62] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/SIGSOFT FSE*, pages 11–20, 2005.

[63] Martin P Robillard and Yam B Chhetri. Recommending reference api documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[64] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[65] Luis R. Rodriguez and Julia Lawall. Increasing automation in the backporting of Linux drivers using Coccinelle. In *11th European Dependable Computing Conference - Dependability in Practice*, 2015.

[66] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE*, pages 499–510, 2007.

[67] Zachary M. Saul, Vladimir Filkov, Premkumar T. Devanbu, and Christian Bird. Recommending random walks. In *ESEC/SIGSOFT FSE*, pages 15–24, 2007.

[68] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *IEEE Trans. Software Eng.*, 2008.

[69] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. Artificial Intellegence*, 2009, 2009.

[70] C. Sun, D. Lo, X. Wang, J. Jiang, and S-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *ICSE*, pages 45–56, 2010.

[71] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, pages 253–262, 2011.

[72] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, 2011.

[73] Laszlo Szathmary, Amedeo Napoli, and Sergei O. Kuznetsov. Zart: A multi-functional itemset mining algorithm. In *CLA*, 2007.

[74] Loren Terveen and Will Hill. Beyond recommender systems: Helping people help each other. In *HCI in the New Millennium*, 2001.

[75] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213, 2007.

[76] F. Thung, S. Wang, D. Lo, and J. Lawall. Automatic recommendation of API methods from feature requests. In *ASE*, 2013.

[77] Ferdian Thung, LO David, and Julia Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE 2013): Proceedings: Koblenz, Germany, 14-17 October 2013*, pages 182–191, 2013.

[78] Ferdian Thung, Xuan-Bach D Le, David Lo, and Julia Lawall. Recommending code changes for automatic backporting of linux device drivers. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 222–232. IEEE, 2016.

[79] Yuan Tian, Dinusha Wijedasa, David Lo, and Claire Le Gouesy. Learning to rank for bug report assignee recommendation. In *IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[80] Shaowei Wang, David Lo, Zhenchang Xing, and Lingxiao Jiang. Concern localization using information retrieval: An empirical study on linux kernel. In *WCRE*, pages 92–96, 2011.

[81] Xiaoyin Wang, David Lo, Jing Jiang, Lu Zhang, and Hong Mei. Extracting paraphrases of technical terms from noisy parallel software corpora. In *ACL/IJCNLP*, 2009.

[82] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470, 2008.

[83] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 325–334. ACM, 2010.

[84] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Accurate developer recommendation for bug resolution. In *20th working conference on Reverse engineering (WCRE)*, pages 72–81. IEEE, 2013.

[85] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *IEEE 27th International Software Reliability Engineering (ISSRE) Symposium on*, pages 127–137. IEEE, 2016.

[86] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699. ACM, 2014.

[87] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge 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.

[88] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, pages 826–836. IEEE Press, 2012.

[89] Qirun Zhang, Wujie Zheng, and Michael R. Lyu. Flow-augmented call graph: A new foundation for taming api complexity. In *FASE*, pages 386–400, 2011.

[90] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static noninteractive approach to feature location. In *TOSEM*, 2006.

[91] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, pages 318–343, 2009.

[92] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.