4-2021

API Recommendation Using Domain And Source Code Knowledge

Rana Kareem Talib Al-Rubaye
ra9118@rit.edu

API Recommendation Using Domain And Source Code Knowledge

by

Rana Kareem Talib Al-Rubaye

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Software Engineering

Department of Software Engineering
B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
April, 2021

API Recommendation Using Domain And Source Code Knowledge

by

Rana Kareem Talib Al-Rubaye

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements of the degree of Master in Software Engineering.

J Scott Hawker — Date
SE Graduate Program Director

Mohamed Wiem Mkaouer — Date
Advisor

Christian Newman — Date
Committee Member

API Recommendation Using Domain And Source Code Knowledge

by

Rana Kareem Talib Al-Rubaye

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Master Program in Software Engineering
in partial fulfillment of the requirements for the
Master Degree
at the Rochester Institute of Technology

Abstract

The process of migration the old retired API(Application Programming Interface) with new and most to up to date one, know as API migration. Developers need to fully understand the documentation for the retired (replaced) library and the new (replacing) library to do the appropriate migration. This manual process is complex, error-prone, and costly for companies. There have been many studies focused on the automation recommendation between different method mapping for different libraries. These studies focused on the recommendations between methods from different programming languages while non of them focused on the recommendations between methods of libraries that belong to the same programming language. At times, one of the studies indicates automatic recommendation when mapping two different methods libraries that belong to the same programming language by using domain knowledge(method description, method parameters|name). In this thesis, we investigated the mapping between two methods of library migrations by using the **domain knowledge** and **source code documentation**. In order to be able to obtain these scenarios, we propose the RAPIM++ machine learning approach which recommends a correct mapping between source and target methods of three-party libraries using domain knowledge and source code knowledge. Our main contribution in this study was, build a model which depends on existing library changes done manually from previous developers in different open source projects in java programming language then use features related to source code implementation, the similarity between method signatures and methods documentation to predict correct method mapping between two methods level library migration. Our result was RAPIM++ was able to successfully mapping between two methods from different third-party libraries with the rate

of accuracy score 84.4%. Additionally, our approach could able to recommend the libraries that absent the documentations since it relies on the source code knowledge along with the main knowledge. We can conclude from these results that RAPIM++ able to recommend third-party libraries with or without documentation, so though libraries that are not well known and do not belong to popular frameworks, can find comprehensive recommendations when using our model. Furthermore, RAPIM++ provides the research and industry community with a lightweight web service that available publicly to make method mapping between third - part libraries an easy task for developers.

Acknowledgments

Throughout my research and study period, many people helped and encouraged me to achieve this milestone. First,I would like to express my deep sense of thanking my mentor and my soul my husband **Dr. Hussein Al-rubaye** to his support and encouragement because without him this mission was impossible to become a reality.

Next, It is my pleasure to thank my advisor **Dr. Mohamed Wiem Mkaouer** for his scholarly advice and knowledge-based approaches which helped me to accomplish this study. And I would like to thank the software engineering department and **Dr.J Scott Hawker** for providing excellent resources and kind assistance during my study journey.

Next, I would like to thank my four little girls **Jena, Laya, Dora, and Yasmine** to inspire me in my all study journey and motivate me to do my best in order to make them happy and proud. Wishing to achieve my entire life goal and be such a good role model for them in their life journey.

Finally, I would like to thank my Mom and my Dad **Zahrah** and **Kareem** to motivate me in every phone call and never forget me in their prayer. Thank you for my all family and love you all.

Thanks,
Rana,

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

1.1 Introduction

Today, software developers rely heavily on third-party liberties for software maintenance purposes due to not consuming time on implementation and produce high-quality software which has up-to-date features. This maintenance process for the software expanding up to 70% of a given software product life-cycle[1]. The process of replacing the old version of a retired third-party library with a new third-party library while keeping the same behavior for the code is called library migration[2]. The migration process between libraries used to be time-consuming, error-prone, and difficult, especially for developers who have not enough knowledge of coding skills. In the same context, one of the previous studies showed that the developers should have at least five years of experience to assigned to migration processes tasks to avoid low performance in the migration process [3]. For that reason, previous studies showed that the companies rely on developers with high coding skills when it comes to migration [3] to avoid any possibility of not delivering high-quality performance, which might be a costly situation for some of the companies. Also, another study shows, developers spend up to 42 days migrating from one third-party library to another [4].

There are a number of recent studies proposed of identifying techniques and processes in order to better replacement the obsolete library with the recent up to date version of the

same API that keeps the same behavior [1,2,3,4]. There are other studies that have attempted to use library recommendation, where they choose the most appropriate adapted library with the obsolete one [5,6,7]. While the previous studies did a good work with recommending the same library method between two different languages, non of them focuses on the need for an inclusive techniques at method level recommendation which takes as an input two different libraries and provide the possible mapping between them and how we replace the new version with the retried one at method level [8,9,10].

The state of the art RAPIM [11] was able building that niche by providing an automatically tool which be able to do the potential mapping at method level between two different third party libraries. RAPIM (Recommending API Migrations) [11] uses and take an advantage of pre-defined features related to the domain knowledge which the similarity between the method signatures and method textual documentation. This features has defined manually and they provide the appropriate recommendations mapping at API method level. from the developers. The main idea behind RAPIM [11] is to take as input two migration libraries, and provide as an output the appropriate mapping between there API methods.

Although RAPIM [11] build a novel machine learning model, that utilized the *Domain Knowledge* and learn from the API method signatures and its corresponding documentations, there is still the need to more global approach, which will be able to use the *Domain Knowledge* (the similarity between the source code and its corresponding documentations) and the *Source Code Knowledge*. Our study RAPIM++ will extend the state of the art study RAPIM [11] and fulling the gap of the current study.

1.1.1 Contribution

Our main contribution in this study we summarized in three steps:

1.We conducted RAPIM++ a machine learning technique, working automatically for library migration purposes. The basic idea behind RAPIM++, it takes two inputs at the API method level, which is the domain knowledge and the source code knowledge along with their method signatures and documentation and source code implementation and

delivers as an output the corresponding mapping between two API libraries. RAPIM++ learns and takes advantage of previous migrations that were reported manually from the developers. RAPIM++ builds its model depending on the numerous features like domain knowledge and source code knowledge.

2. To evaluate the performance of our study RAPIM++ in detecting the correct mapping between different API libraries, we conducted an empirical study on 8 popular migrations. The result indicates that RAPIM ++ was able to detect the correct mapping between libraries and perform in an efficient way in comparing to the state-of-the-art technique RAPIM. Despite RAPIM showed a high percentage of accuracy 80.56% when it uses just the domain knowledge, RAPIM++ was able to increase the accuracy to 84.4% where it uses both domain knowledge and source code knowledge along with their documentation for method level API library migrations.

3. We provide an easy to access web service which include all the implementation and the data set for the RAPIM++ . In this case, we provide the community of researchers and developers a good free resource to take an advantage from it or improve it.

1.2 Background and Terminology

In this section, we are going to provide background information for all algorithms that current studies rely on. This will help to understand the algorithms that been used in coming chapters and how these algorithms work. In addition, in this section we will give brief introduction for main concepts that we will use in this study

1.2.1 Library

Library is a combination of the non volatile resources. Can accesses it easily since it is available for a public use over library's Application Programming Interface(API) used by general purpose languages for write high level program so instead of request the server over and over, we can use a library to give high performance implementation. However, the library is set of behaviour which structured to be able to reuse it by different computer

programs for software development. These computer programs will call the library in different mechanism depend on the structure for each programming language. Just like any classic software, API has many versions. Each version consider as the upgrading of the old one and it has the enhancing features.

1.2.2 Library Migration

Library migration is the process of exchanging the existing library with the new one without changing the code performance.The existing library or the old library Will consider as a retired in two cases. One:if the liberties that depends on the existing library are not updated. Second: or when there are competition between two libraries, then the library which appear in better performance or which has more feature will win in this competition and will consider the other one as a old library. A migration process occurs when a new library is replacing a retired library. The retired library is considered retired if all of functions dependencies are removed from the project source code. As shown in Figure 1.1 developer migration *json* version *20090211* to *gson* version *2.2.4* in the *java_mega_api*[1].

```
-       <groupId>org.json</groupId>
-       <artifactId>json</artifactId>
-       <version>20090211</version>
+       <groupId>com.google.code.gson</groupId>
+       <artifactId>gson</artifactId>
+       <version>2.2.4</version>
```

Figure 1.1: Sample of dependency file during Library migration from *json* and *gson*.

[1]https://github.com/danbrough/java_mega_api/commit/cb057c4696d95a9a9da99eb4cf88f60aaf0e9ee2

1.2.3 Migration Example

In this section we will explain an example for migration at method-level. Figure 1.2 elaborate one of the popular library migration rule process when we replacing the library *json* with the *gson* library[2]. The process of changing group of source codes from removed library with group of source code changes in target library, it called *Migration fragment* [12]. The Figure 1.2 below explains two *Migration fragment*: **First Fragment**, we can observe one mapping, where the class named *JSONObject* replaced with class named *JsonObject*, while two methods they have same named but different capitalization letters. **Second Fragment**, we have two mappings, in the first mapping, The method *JSONException*(*string*) has been replaced with methods, namely *RuntimeException*(*string*). In the second mapping, The method *getLong*(*string*) has been replaced with two methods, namely *get*(*string*).

```
   @Override
1  public void onResponse(Object o) throws JSONException {
-    JSONObject job = (JSONObject) o;
+  public void onResponse(Object o) {
+    JsonObject job = (JsonObject) o;
     if (!job.has("mstrg"))
2      throw new JSONException("Expecting a mstrg in the response");
-    quota = job.getLong("mstrg");
+      throw new RuntimeException("Expecting a mstrg in the response");
+    quota = job.get("mstrg").getAsLong();
   }
```

Figure 1.2: Sample of migration between *json* and *gson*.

1.2.4 Library Upgrade

For the refactoring purpose, developers may attempting library upgrade. There are different goals for library upgrade, it could be for adding new classes and methods, updating the functionality for the existing system, or adding new classes to the existing packages.

[2]https://github.com/danbrough/java_mega_api/commit/cb057c4696d95a9a9da99eb4cf88f60aaf0e9ee2

Deprecating methods or classes during the upgrade process, may has dis advantage for the API since its prevent API developers from the evaluation because they are restricted by the old version of the API. All the modification that the upgrading makes should not effect the utilization in the client software side.

1.2.5 Migration Rule

A migration rule represent by a pair of a *source* (retired) library and a *target* (replacing) library, *i.e.*, $source \rightarrow target$. For example, $json \rightarrow gson$ represent a migration rule where the library json[3] is migrated to the new library gson[4].

1.2.6 Method Mapping.

Source library has listed methods that have been used in code, and the target library has a list of methods that need to be used in the code. The process of mapping the methods from source library to method from target library called *Method Mapping*. Given method from source library may be mapped to zero or many methods in the target library. Also, a method from the target library can be a target for zero or many methods from the source library. Zero means that methods don't have a mapping with another method from source or target.

1.2.7 Text Reprocessing (TR)

Text processing is one of the popular tasks in machine learning applications. The goal of text processing is that it could help to increase the accuracy of the Natural Language Processing (NLP) task when we use it correctly. There are different examples where text processing used in machine learning applications, for instance it could be used filtering the spam messages by detecting unwanted messages and move it to spam organization.The way that we use the text processing is for filtering unwanted words and characters from

[3]https://www.json.org/
[4]https://github.com/google/gson

the method description. Let consider d as a description for the library at the method level. d has undesirable words like "is", "the" and characters like dot and colon. We use Nature Language Processing to remove this words and characters in order to get method description with reducing the noises when trying to calculate the similarity between two methods and get high accuracy.

$$\widehat{d} = TR(d) \tag{1.1}$$

1.2.8 Vector Space Representation

In our methodology, we want to find patterns between the description (documentation) of source and target libraries, and the measure of these data can be related to each either. How we can generate features from these text data. To do so, we need to find the similarity between the description of methods, descriptions of parameters, descriptions of return types. All these data are text data. To generate features from these data, we need to be converted text to numeric data, then apply mathematics equations such as to measure similarity score and consider this score as a feature.

To convert text (sentence) to numeric, we use *Vector Space Representation*. This representation generates the weight vector W_d, which is a vector of numeric numbers for given words in a sentence. This vector has weight for every word in given sentence $w_{t,d}$. We calculated this weight using *Frequency-Inverse Document Frequency* (*TF-IDF*) as shown in equation 1.2. The weight for every word $w_{t,d}$ calculates by dividing the number of times word appear in sentence $tf_{t,d}$ by the number of words in a sentence. To reduce the noise of word that appears many times in document such as (is, are), This data get to multiply by *log* for the number of documents N which her number of methods descriptions divided by the number of document df_t were given the word appear.

$$W_d = \begin{bmatrix} w_{t_1,d} \\ w_{t_2,d} \\ \vdots \\ w_{t_n,d} \end{bmatrix}, w_{t,d} = \frac{tf_{t,d}}{t_n} * log\left(\frac{N}{df_t}\right) \tag{1.2}$$

Now we have weight vector W_d for a description of source/target methods, the weight vector W_d for descriptions of parameters for source/target methods, the weight vector W_d for descriptions of return types source/target methods. We use cosine similarity $cos(s,t)$ to calculate the similarity score between given two vectors and consider it as a feature as shown in equation 1.3. For example, we calculate cosine similarity between weight vector W_d for source method description and weight vector W_d for target method description that will generate a single score we consider that score as a feature. The same method applied for measuring cosine similarity between weight vectors W_d for descriptions of source/target methods parameters, cosine similarity between weight vectors W_d for descriptions of source/target methods return types.

$$cos(s,t) = \frac{W_s \cdot W_t}{||W_s|| \cdot ||W_t||} \tag{1.3}$$

1.2.9 Words Extraction (WE).

Let d be string has a number of words combined, for example, combined names in the method signature, such as we have a method named *getAsLog* this just sentence *"get as long"*. We can extract this type of sentence then apply text processing. Also combined names in package import, for example *org.apache.maven.enforcer* which is *"org apache maven enforcer"*. In this step, we extract d^* which represent words in a string, using the function that we named Words Extraction WE as follows:

$$d^* = WE(d) \tag{1.4}$$

For example, if d is method signature in *easymock*[5], then d^* is generated using WE which is described as follows:

[5]https://github.com/apache/maven-enforcer/commit/12b3260071b94f66c078ca4bfef07fe8d28fdea7

Words Extraction (WE)

input (d): *'evalControl.expectAndDefaultThrow'*.
1- Special Characters Cleanup: In this step, we search for dots and replace then with spaces. If we apply this approach on the current example, the output for this step is $'evalControl < space > expectAndDefaultThrow'$.
2- Camel Case Splitter: In this step, find all words in a given string by splitting the string base on a camel case. We assume every new word in a combined string start with a capital letter The output for this step is $'eval < space > Control < space > expect < space > And < space > Default < space > Throw'$. We know different languages using different coding standard, but we used java, and camel case is very common in java code..
Output(d^*)**:** *'eval Control expect And Default Throw'*

1.2.10 Machine Learning

machine Learning is the process to make the computer to learn and advance from the previous data set an automatically without being explicitly programmed and it is a subset of AI(Artificial Intelligence). Training model is a sample data which we can use it to build our machine learning model on it. The goal of the training model is that to make prediction or decisions. There are various approach that the discipline machine learning uses in order to complete the tasks that may not specific algorithm available to finish it. One of this approach is labeling the data set manually then we can name it training data set. The goal of the training data set is that make the model learn from the training data set in order to improve the performance of the algorithm to determine the correct answers. There are three approaches for machine learning depending on the feedback that learner provided: unsupervised machine learning, supervised machine learning and reinforcement learning. One example of machine learning is email filtering, when we want to filters the emails as a spam and not spam we should manually execute which the spam and not spam emails is and write tons of if statements which will turn out by not effective way to solve this particular problem,in this way we can use machine learning algorithms, so the goal of machine learning algorithms are solve problems which are infeasible being solve it by other way.

Model.

There are two important terms in machine learning which are algorithm and model. Algorithm is a series of steps which we can execute it in code and run it in data set. However the model is the outcome(output) of the algorithm and it consist of the model data and the algorithm that responsible on the prediction. In other word a model is a representation of what was detected by the algorithm which can deliver kind of an automatic programming. So a training data set is a model which we can use it to predict data that has not seen before to make decisions whether our real model predict correctly or not.

Model tuning.

The main goal of doing model tuning for our model is to maximise the performance of our model in same time avoid the over fitting and too high differences. The tuning process can be done by changing the hyper parameters for instance the number of trees based algorithms or the number of values in linear based algorithms, re run the algorithms on the data set that we have again, finally in order to determine which model is an accurate one, we make comparison between the model performance and the validation data set that we have.

Feature selection.

Feature selection is the procedure that selecting the most relevant features(variables, attributes) from the data set that we have to develop the predictive model and avoiding the irrelevant or redundant from the data set. there are many purpose of using feature selection, one of them is that avoid the over fitting by customizing the data and enhance it. Most of the data set that using in any experiment the data could contain redundant or irrelevant features,So the goal here, how we can use remove the redundancy without removing the relevant information from the data set. Using feature selection technique will do this job

1.2.11 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) [13] is one of the most valuable approach which can analyze the data statistically. PCA can compact and convert the m numbers of data by m numbers of data. It eliminate the potential numbers of data set and keep the same value of information at same time. The goal of using PCA is to reduce the complexity and simplify the structure of the data set. In our study PCA take very important role when we extract features from the source code knowledge. After we use Code2vec [14] we will extraxt a huge numbers of the features. Then we will feed this major numbers of feature to the PCA algorithm in order to avoid over-fitting issues and make the model run faster.

Recurrent neural network(RNN)

Recurrent Neural Network(RNN) is the a machine learning algorithm which intended to take a series of input, which the size of input is not predetermined and provide the series of output. One or more input could provide one or more output, where the input could be effected by by the hidden state vector which we can update it and use it as the input for the next output in the series. Each input has relationship with others and its influenced by the other inputs. The basic idea of how (RNN) working is that the (RNN) algorithm learn from the past and its decision is depending and influenced by the past. Its learning from the first look from the training data set then using this knowledge to make an appropriate generation in the output.

1.2.12 Method-level Vector space representation.

The main idea behind the code2vec [14] is that takes data already labeled it at method level and convert it to vector. The main purpose of this step is that to represent a method code source implementation as a vectors

Chapter 2

Literature Review

2.1 Abstract

This section will provide a number of related literature that describes the migration process between different API and how the machine learning process plays an important role in facilitating the migration from one library to another.

2.2 Popular API recommendation techniques

It is very important to understand *how* the migration process can occur between different APIs. There are many studies have been discussed a numbers of tools and techniques in order to facilities the migration process between different APIs. Most of theses research studies took advantage of the documentations for the API to do more appropriate recommendation. One of these studies focused on the code examples between deprecated and the new API [15]. In this paper Lamothe et al. attempted to use open source repose tries for API android applications written in java to recommend APIs mapping. This study propose A3 tool that can automatically generate API migration patterns from code example from public repositories.This tool was successfully able to generate API migration patterns with ratio 96.7% precision. Another study focused on mapping between

deprecated and target APIs by using heuristics and source code analysis [16]. This study proposed NEAT which is automatic tool to recommend mapping between the library migrations APIs without using th code change examples. This study assumes the availability for APIS documentations and it uses the android APIs that available publicly. Also giving the edits with replacing API was interesting topic for previous work [17]. This study suggested the API migration mapping between old version of API with new release and up to date one along with the edits with the new version of API . Although, this automation approach provide list with all edits and suggest the most appropriate one to the client was good attempts, but it edit can not handle big changes like recommend change the concrete class with abstract class. We can obvious from previous work that most of the API library migration techniques relies heavily on the documentations which might not well written from some developers, despite of popular APIs and framework include all appropriate documentations.

2.3 API mapping between two different programming languages

Different studies focused on the API library migration mapping between different programming languages at the same API. Pandita et al. [9] suggested the recommendation mapping same API by using different programming languages which is Java and C#. The goal of this paper is detecting the method mapping between the source and target library by automatically locating the potential method mapping between their APIs.All the techniques applied by using text mining on the functions textual descriptions. This work has been extended to involve more classes and include C# language and an android projects [10]. Although previous work focused on the static analysis and the corresponding mapping between the method between two different library across the same API, the dynamic analysis has been the focus for the researcher in their work [8]. In this paper the authors focused on a developing strategy to develop relevant method mapping between Java2 mobile edition and an Android graphics. Object Oriented style standard was an interesting topic for many researchers. Martinez et al. [18] followed in his research the style of object oriented programming standard as a general and the migration process between

programming languages C|C++ and mobile applications in specifically.

2.4 API mapping at method level

Xu et al. [19] propose MULAPI which recommends the API mapping at method level by using description and historical use for API and where has been used. MULAPI takes into account the information stored in the codebase and add one API usage location component for API recommendation.

Haung et al. [20] present BIKER an automatic techniques uses stack overflow posts and API documentation to give an appropriate API mapping at class and method level. This study leverage the gap between API documentation which might missing the relevant information and stack overflow posts that describe all the material to use the most relevant API in programming tasks. This study uses word embedding methods in order to convert all the sentences in both API documentation and SO posts to vectors and find the relevant similarities between them. This research was good step to right direction. The gap of this study is that it uses just the description feature of API to recommend API mapping at class and method level.

Alrubaye et al. [11] propose RAPIM primary inspiration in our work, which is a machine learning technique that recommends mapping between two different libraries at the method level. The finding for this study is that RAPIM was successfully able to recommend the mapping between two unknown libraries by extracting an important features from the lexical similarity from the *domain knowledge* for methods along with the documentation for the libraries. RAPIM was able to recommended correct mapping and score good ratio of accuracy 87%. Furthermore, Other studies by Alrubaye [12, 21, 22, 23, 24, 25] have studied the library migration at method-level as well.

Chapter 3

Methodology

3.1 An Approach Overview

The main idea of our approach RAPIM++ is try to take an advantage from the migration mapping that done manually in different open source projects from previous developers and try to reuse the past experience.In this chapter we will go over our approach overview, then we will explain in detail all the methodology in a different steps. Migration rule means that the migrating process between a pair of libraries which is the source library (Removed) and the target library (replaced). Let say L_s is the source library and the L_t is the target library. The representation for the the process for the migration rule will be like: $L_s \rightarrow L_t$. There are many of open source projects on GitHub that represent the migration rule, one of the popular which represent the migration rule is *easymock* $\rightarrow$ *mockito* . For a migration rule where L_s is a source method let consider it an equivalent for a M_s where $m_s^{(i)} = \{m_1, m_2, ..., m_{L_s}\}$, , and L_t is a target method where a L_t is an equivalent to M_t and the $m_t^{(i)} = \{m_1, m_2, ..., m_{L_t}\}$. Our main goal is that to find an appropriate mapping between both L_s and L_t.

$$f : L_s \rightarrow L_t \tag{3.1}$$

The process of mapping the source method $m_s^{(i)} \in L_s$ with the equivalent target method $m_t^{(i)} \in L_t$ we call it *Method Mapping*. In our approach,we used Microsoft azure machine learning studio in order to generate our models. We use this platform because it has all the

pre-build machine learning techniques, which not required us to build all the algorithms from the scratches. This property saved us a lot of time. Also it is easy to access from different users

Figure 3.1 gives an overview of our approach, which made the collaboration done with an efficient way. , which consist of the collection of two main feature stages: the first stage is: *Domain knowledge*: in this phase, we looking for the domain knowledge, which consist of the significant information that gathered from the method signatures and library documentations. To extract all the features which we need to build our model, we used the data set [12], which contain all the library mapping. There are different steps to complete the domain knowledge stage: First, we collect all the APIs from the data set [12] along with its matching documentation; Second, we do the text prepossessing for documentation. The main goal of this step is that to reduce the noise when we calculating the similarity value between the source/target documentation; Third, we do feature engineering for the features $\varphi_1 ... \varphi_8$, the goal of this step is that to generate numeric features from the text features that we have, in order to gain more accurate result. The second stage of our approach is that *Source code Knowledge*: There are different steps for this phase: First, from the data set that we have [12] which include all the mapping, we gathered all the APIs with its matching methods source code implementation; Second, we need to build a model which be able to predict the similarity score (feature φ_9) between the given code vectors of the Source/target methods, for this purpose, we use the CNN learner. This model trained to learn from the J2EE interfaces implementation. Third, to decrease the number of features from 384 * 2 to five features which is $\varphi_9 ... \varphi_{13}$, we need to apply Principal Component Analysis(PCA). The purpose of this step is that to avoid the over fitting issues, since we have a huge numbers of features as an input. So reducing the number of input to five only will reduce the noise and makes the model run faster. In order to build our recommendation model, we use all the mentioned features alongside with output class passed.

3.2 Domain Knowledge

In This section we describe how we collect and extract features ($\varphi_1 ... \varphi_8$) from domain knowledge (library documentation, and method signatures).

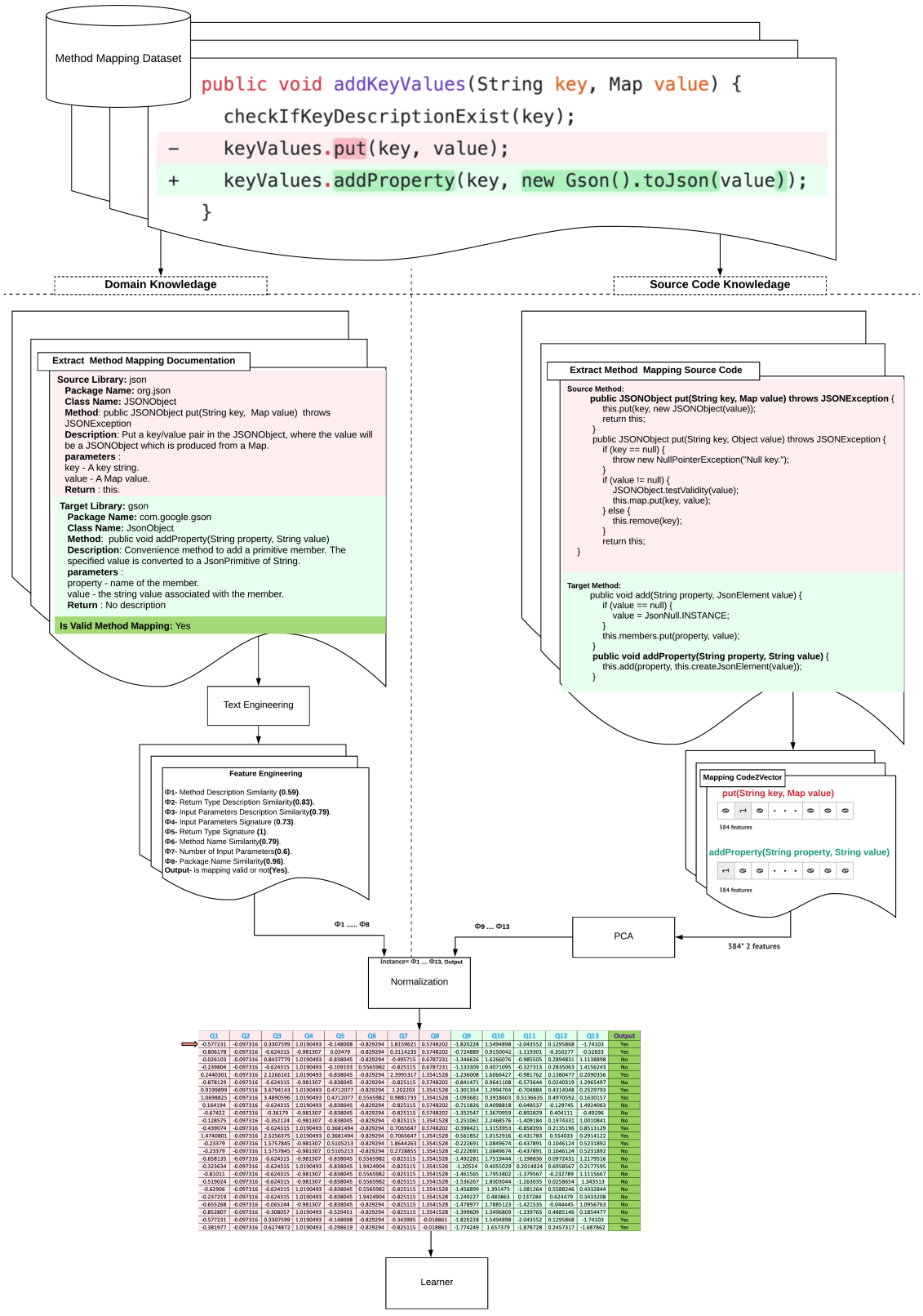


Figure 3.1: The proposed RAPIM++ approach for method-level recommendation.

3.2.1 Extract Method Mapping Documentation

In this phase, we collect library documentation for given method mapping in alrubaye dataset [12]. As show in Figure 3.1 where we collection documentation for following methods $JSONObject\ put(String\ key, Map\ value)$, from *json*, to the method $void\ addProperty(String\ property, String\ value)$, offered by *gson*. Thanks for Alrubaye [11] study that offer *Documentation Collector* that we used to collects documentation for both the source methods and the target methods. Given a migration rule as input to the *Documentation Collector*. The tool collects the method's documentation associated with all methods in source/target libraries. Then we search for documentation of methods that involve method mapping.

Our goal in this section is generating eight different features $\varphi_1(s,t)$ to $\varphi_8(s,t)$ for source (s) and target (t) method's documentation and signatures. To do so, first, we need to convert this documentation from text data to numeric data. We discuss in the background how we can do that using *Vector Space Presentation* with cosine similarity. Next, We will discuss in detail next how we generate every feature one by one.

3.2.2 Feature Engineering

In this section, we describe how we extract every feature from domain knowledge.

Method Description Score φ_1

In this section, we want to extract feature $\varphi_1(s,t)$ which represents a score of how similar the source method description md_s to the target method description md_t. To extract this feature, first, we just measure the cosine similarity between the source method description md_s with the target method description md_t without any prepossessing TR as shown below. We find that if we apply TR on the text before applying cosine similarity will lead to noise and reduce the accuracy by 3%.

$$\varphi_1(s,t) = cos(md_s, md_t) \tag{3.2}$$

For example, if we measure $\varphi_1(s,t)$ for the example in Figure 3.1, we apply the cosine similarity between md_s (*"Put a key/value pair in the JSONObject, where the value will be a JSONObject which is produced from a Map."*), and md_t (*"Convenience method to add a primitive member. The specified value is converted to a JsonPrimitive of String."*). The output will be a similarity score of (0.52).

Return Type Description Score φ_2

In this section, we want to extract feature $\varphi_2(s,t)$ which represents a score of how similar the source method return type description rtd_s to the target method return type description rtd_t. To extract this feature, first, we apply text reprocessing TR on the source/target method return type description to clean sentences beside description. That will generate $\widehat{rtd_s}$ from source method return type description and $\widehat{rtd_t}$ target method return type description. then we just measure the cosine similarity between $\widehat{rtd_s}$ and $\widehat{rtd_t}$.

$$\varphi_2(s,t) = cos(\widehat{rtd_S}, \widehat{rtd_t}) \tag{3.3}$$

For example, to measure $\varphi_2(s,t)$ from the example in Figure 3.1, we apply TR on both rtd_s (*" this. "*) and rtd_t (*"No description*) to get $\widehat{rtd_s}$ and $\widehat{rtd_t}$. We then apply the cosine similarity between $\widehat{rtd_s}$ and $\widehat{rtd_t}$. The output will be a similarity score of (0.81).

Input Parameters Description Score φ_3

In this section, we want to extract feature $\varphi_3(s,t)$ which represents a score of how similar the source method input parameters description ipd_s to the target method input parameters description ipd_t. To extract this feature, first, we apply text reprocessing TR on the source/target method input parameters description to clean sentences beside description. That will generate $\widehat{ipd_s}$ from source method input parameters description and $\widehat{ipd_t}$ target

method input parameters description. then we just measure the cosine similarity between $\widehat{ipd_s}$ and $\widehat{ipd_t}$.

$$\varphi_3(s,t) = cos(\widehat{ipd_s}, \widehat{ipd_t}) \tag{3.4}$$

For example, to measure $\varphi_3(s,t)$ from the example in Figure 3.1, we apply TR on both ipd_s (*"key - A key string. | value - A Map value."*), and ipd_t (*"property - name of the member.| value - the string value associated with the member."*) to get $\widehat{ipd_s}$, and $\widehat{ipd_t}$, then We measure the cosine similarity between $\widehat{ipd_s}$, and $\widehat{ipd_t}$. IThe output will be a similarity score of(0.75).

Input Parameters Signature Score φ_4

In this section, we want to extract feature $\varphi_4(s,t)$ which represents a score of how similar the source method input parameters ips_s to the target method input parameters ips_t. To extract this feature, first, we apply word extraction WE on the source/target method input parameters to extract words from parameter names. That will generate ips_s^* from source method input parameters and ips_t^* target method input parameters. then we just measure the cosine similarity between ips_s^* and ips_t^*.

$$\varphi_4(s,t) = cos(ips_s^*, ips_t^*) \tag{3.5}$$

For example, to measure $\varphi_4(s,t)$ from the example in Figure 3.1, we apply word extraction WE on both ips_s (*"String key, Map value"*), and ips_t(*"String property, String value"*) to get ips_s^*, and ips_t^*, then We apply the cosine similarity between ips_s^*, and ips_t^*. The output will be a similarity score of(0.71).

Return Type Signature Score φ_5

In this section, we want to extract feature $\varphi_5(s,t)$ which represents a score of how similar the source method return type signature rts_s to the target method return type signature rts_t. To extract this feature, we just compare if the return types of source/target methods

are the same we return one otherwise we return zero, as shown below.

$$\varphi_5(s,t) = \begin{cases} 1 & \text{if } rts_s \text{ is equal to } rts_t \\ 0 & \text{if } rts_s \text{ is not equal to } rts_t \end{cases} \tag{3.6}$$

For example, to measure $\varphi_5(s,t)$ for example in Figure 3.1, both rts_s return *JSONObject*, while rts_t returns *JSONObject*, The output will be (0).

Method Name Score φ_6

In this section, we want to extract feature $\varphi_6(s,t)$ which represents a score of how similar the source method name $methodName_s$ to the target method name $methodName_t$. To extract this feature, first, we apply word extraction WE on the source/target method name to extract words from names. That will generate $methodName_s^*$ from source method name and $methodName_t^*$ target method name. then we just measure the cosine similarity between $methodName_s^*$ and $methodName_t^*$.

$$\varphi_6(s,t) = cos(methodName_s^*, methodName_t^*) \tag{3.7}$$

For example, to measure $\varphi_6(s,t)$ from the example in Figure 3.1, we apply WE on both $methodName_s$ (*"put"*), and $methodName_t$ (*"addProperty"*) to get $methodName_s^*$, and $methodName_t^*$, then We measure the cosine similarity between $methodName_s^*$, and $methodName_t^*$. The output will be a similarity score of (0.79).

Number of Input Parameters Score φ_7

In this section, we want to extract feature $\varphi_7(s,t)$ which represents a score of how similar the number of parameters in source method $inputParamCount_s$ to the number of parameters in target method name $inputParamCount_t$. To extract this feature, we apply the following equation.

$$\varphi_7(s,t) = 1 - \frac{|inputParamCount_s - inputParamCount_t|}{inputParamCount_s + inputParamCount_t} \tag{3.8}$$

For example, to measure $\varphi_7(s,t)$ from the example in Figure 3.1, we find different between $inputParamCount_s$ which has two parameters which are *key,and value*, and $inputParamCount_t$ that has two input parameters which is (*property,and value*), so The output will be a similarity score of (1).

Package Name Score φ_8

In this section, we want to extract feature $\varphi_8(s,t)$ which represents a score of how similar the source method package name $packageName_s$ to the target method package name $packageName_t$. To extract this feature, first, we apply word extraction WE on the source/-target method package name to extract words from package names. That will generate $packageName_s^*$ from source method package name and $packageName_t^*$ from target method package name. then we just measure the cosine similarity between $packageName_s^*$ and $packageName_t^*$.

$$\varphi_8(s,t) = cos(packageName_s^*, packageName_t^*) \tag{3.9}$$

For example, to measure $\varphi_8(s,t)$ from the example in Figure 3.1, we apply word extraction WE on both $packageName_s$ (*"org.json"*), and $packageName_t$ (*"com.google.gson"*) to get $packageName_s^*$ which is (*"org json"*), and $packageName_t^*$ which is (*"com google gson"*), then We measure the cosine similarity between $packageName_s^*$, and $packageName_t^*$. The output will be a similarity score of (0.96).

3.3 Source Code Knowledge

In This section, we describe how we collect and extract features $(\varphi_9 ... \varphi_{13})$ from source code knowledge (methods source code implementation).

3.3.1 Extract Method Mapping Source Code

In this phase, we collect method source code implementation for given method mapping in alrubaye dataset [12]. As show in Figure 3.1 where we collection source code implementation for following methods $JSONObject\ put(String\ key, Map\ value)$, from *json*, to the method $void\ addProperty(String\ property, String\ value)$, offered by *gson*. To do so, We first download the source/target third-party libraries *jar* files. Then we reverse engineer *jar* files to source code using *cfr_0_114.jar* [1]. Then we wrote parser to parse all classes and their associated methods. Then for give method mapping we search for source code for both methods in source/target libraries. We notice that there a method is just a wrapper for other methods. These types of wrapping developers avoid breaking changes for projects that upgrade to a new version of API, and their calls for old API methods continue working. For our work, having a wrapper is not be helpful to extract similarity information from source code that is just one line call to another method. Our parser can handle these issues by un-wrapping methods that just wrapper. For example, if method $A()$ implementing has one call $B()$, we just consider code implementing for $B()$ is code implementing for method $A()$ source code. So instead return $B()$ as source code body for method $A()$, we return the code source code body implementation for $B()$ as $A()$ source code body.

3.3.2 Mapping Code2Vector

In this step we apply code2Vec [14] on method source code implementation for given method mapping in alrubaye dataset [12]. As show in Figure 3.1 where we generate code vector form following methods source code implementation for
$JSONObject\ put(String\ key, Map\ value)$, from *json*, and source code implementation for method
$void\ addProperty(String\ property, String\ value)$, offered by *gson*. By end this step, we have 384 features generate for ever method source code body. Since method mapping is pair of methods so we will have 384*2 features generated by this step for given method mapping.

[1] https://github.com/hussien89aa/AdsVulnerablilty/blob/master/cfr_0_114.jar

3.3.3 Principal Component Analysis

The code2Vec [14] generates 384 features from a method source code. Since a migration rule has two methods, so we have 384*2 features generated from source/target methods' source code implementations per method mapping. Having such a huge number of features as input to the machine learning model will make the model slow and lead to much noise that may lead to overfilling issues where the model learns from source code knowledge more than what it learns from domain knowledge. We need to represent these 768 features with a fewer number of features while preserving the same feature contribution to the model. Principal Component Analysis (PCA) is a concept in machine learning that can help us here. It is used when we have a large number of features, and we want to represent them with N Component(features) to reduce the noise and make the model run faster. We feed 768 features to PCA and set up the number of a component to five. PCA can generate five components (features) from 768 features. The new features are $(\varphi_9 ... \varphi_{13})$.

3.4 Classifier Model

As we mentioned earlier, we used azure machine learning in order to build our tool. In machine learning techniques, there are a number of classification algorithms that designed specifically to serve our case to reach better result. One of the previous study conducted that an algorithm which takes the classifier that works on *instances* [26]. We divided our data to training and testing data set. In the training part, we trained our classifier on a set of features besides the output which was manually labeled as a valid (the method mapping is done correctly) and in valid(the method mapping not done correctly). Then we applied normalization to our data set in order to put all the data set in same scale of value and avoid outfitting issue since it makes the model runs slow. We specifically used *z-score* method. After we done from this step, we have our model and it is prepared to use. What we do next is feed our model with data set which has never seen before in order to make sure our model is able to detect unknown data. In all our process of our work where we build our models, we used Microsoft Azure machine learning studio [2].

[2]https://studio.azureml.net/

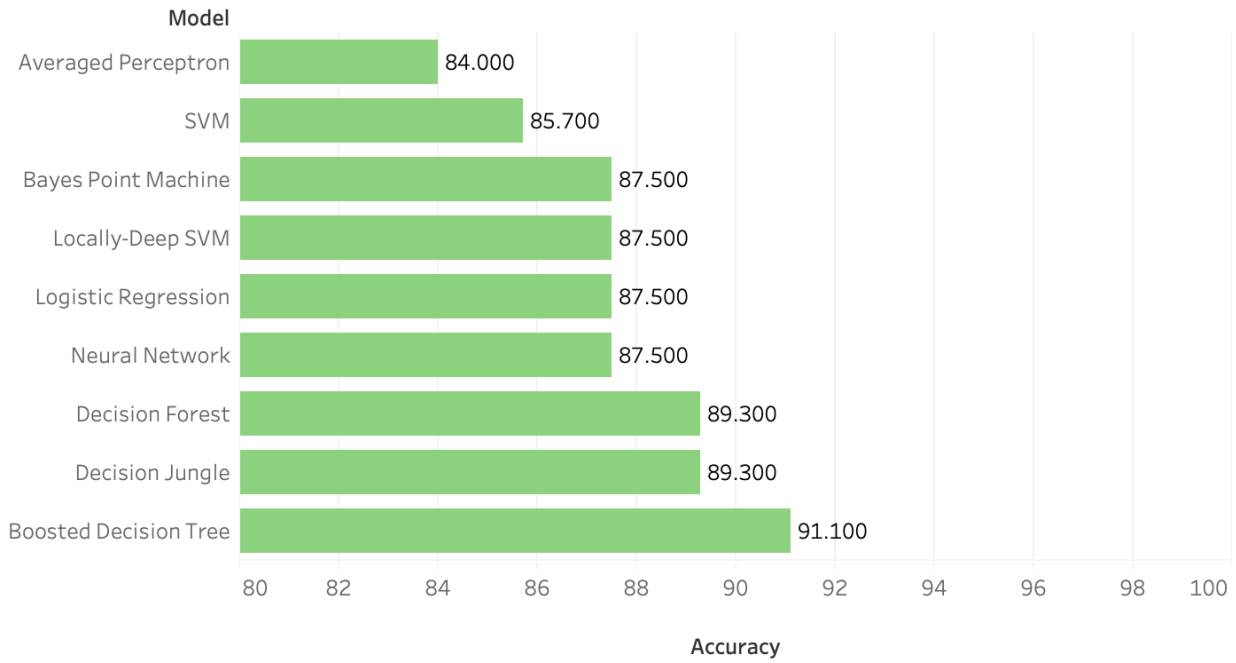


Figure 3.2: Comparative study Different number of features and accuracy.

To make sure our model work perfectly, we conducted an empirical study by trying and running different types of classifiers and make comparison between them then we take the most optimal one.

As obvious in Figure 3.2, we compare different machine learning models on our dataset such as, Random Forest, neural networks, Decision Forest, Decision Jungle,Support Vector Machines (SVM), and Boosted Decision Trees (BDT).from our study, we find that that different algorithms give different accuracy. For example, Neural network is a machine learning model which works as humane brains and take binary inputs and provide binary outputs which we call it perceptron. When we run neural network model classifier, we utilized multi-layer perceptron architecture along with one single hidden layer. To check if this model give use good result, we tested it with different numbers of neurons (we started with five neurons and we ended up with the 50) in the hidden layer. Our result for this study showed that when we increase the number of neurons in the hidden layer, we obtained that the accuracy of testing data set will decrease. The reducing of accuracy

refers to over fitting for the model and the main causes for it lack amount for the data set value. Second, we used Support Vector Machine (SVM) which was wort performance classifier between all. SVM got its best 84.7% accuracy.

Third, we used A Two-Class Boosted Decision Tree (BDT) which is machine learning model belongs to ensemble learning where the new tree correcting from the errors that occurred in first tree. BDT was the optimal and best fit machine learning model for our data set with an accuracy rate of 91.9%. BDTs performed the best specially when trying it on small datasets due to it use of an ensemble of Decision Trees and weighted voting.

Chapter 4

Results

4.1 Abstract

4.2 Experiment Design Setup

In our study we proposed two following research questions:

- **RQ1:** Is RAPIM++ able to generate the correct method mapping? And how we can compare the accuracy of the result to the state of the art approach RAPIM? To give a full answer for this question, we should make sure RAPIM++ will able to recommend the correct method mapping in chosen 8 popular library migrations. We used the same data set [12] of RAPIM in order to justice comparison between RAPIM and RAPIM++. As discussed earlier, RAPIM uses just 8 features generated from domain knowledge while RAPIM++ uses 14 features generated from domain knowledge and source code knowledge along with the output which is a binary classification for the method valid or invalid. Since our training data set is already labeled it, we will use supervised machine learning. We split the data set that we have to training and testing data set as: **1**) We used K fold cross validation model to a achieve the desire result. The numbers of fold we insert it was 9. We choose one of the migration rule for testing and the 8 migrations rule for training set. For example, we used

json→gson for testing and the remaining migrations for training by feeding the all the combination of method mapping as an input to the model in training phase. In the training phase the model lerans how to recommend the correct patterns in both studies RAPIM++ and RAPIM [11]. After our models are trained, we will move to the testing part which is important to make sure our model recommend the correct method mapping across all the third party libraries. We use the migration rule json→gson and provide all the possible method mapping between them and decide if it is valid mapping or in valid mapping. In valid mapping situation means that the mapping between source and target method is done by correct way, while in in valid case means that the mapping between source and target method done by in correct way or there is no mapping between these method. We will repeat all this process over 9 folds in order to better recommendation for any new data to the model. **2**) To make sure RAPIM++ able to estimate the correct mapping between the libraries, we used data set from Teyton et al. [27] done from previous developers manually. Our goal in this step to know weather RAPIM++ able to do the same mapping as done from data set [27] by indicating the correct and incorrect mapping . We challenge the algorithms in RAPIM++ in order to recognize the mapping weather is correct or incorrect as same as the previous developers done manually.
To measure the **Accuracy** we calculated the percentage of all correct method mapping divided by the summation of correct and incorrect mapping. Down below the representation for Accuracy equation:

$$Accuracy = \frac{Tp + Tn}{Tp + Tn + Fp + Fn}$$

Tp(True positive): Is all the numbers of valid method mapping recommended as valid mapping.

Tn(True negative): Is all the number of invalid method mapping recommended as invalid mapping.

Fp(False positive): Is all the number of invalid method mapping recommended as valid mapping.

Fn(False positive): Is all the number of valid method mapping recommended as invalid mapping.

- **RQ2** How can RAPIM++ make better recommendation for method mapping between libraries through lowest loading of training data set?
  To answer this research question, we utilized k-fold Cross validation model. In our specific case, 10 fold cross validation used in order to increase the generation of various combination of mapping rules in each fold. In our first run, we started with one fold training and nine remaining fold for testing after each run we increased the fold for training data size and decreased the fold for testing data size. We ended up by having nine folding size for training and one folding size for testing. The main goal of this research question is that determines the effectiveness of the training model size for RAPIM++ and measure its accuracy comparing to the state of the art approach RAPIM [11].Also, by answering this research question, we will able to perform our solution as a light-weight web service. Furthermore, by evaluating our training data set on existing migrations rule, we will able to generate more solid model.

4.2.1 Method Mapping Implementation

In this section we will describe how we calculated the method mapping between source method L_s, and the target method L_t. For each method we calculated cosine similarity score between L_s, and L_t. And then we return the method that has top matching score when k=1. We have selected k=1 because because for each target method we choose one method.

4.2.2 Learning to Rank (LTR)

Is one of the machine learning approaches which solves ranking problems. The function below will describe the ranking:

$$LTR_{score(s,t)} = \sum_{i=1}^{8} W_i^{LTR} * \varphi_i(s,t) \tag{4.1}$$

The basic idea of LTR is that learns from weight of features through training set. Each feature has its own weight. And its weight will specify its rank . the feature which has

high weight, will has big impact comparing with the feature which has low weight. $\varphi_i(s,t)$ in this part we calculate the relationship between the source method and target method for each feature. We doing this process across eight features which already discussed earlier. However, W_i^{LTR} this part is a result for previous solved mapping from the training data set.

4.2.3 Function Signature (FS)

Function Signature knows as functions for input and output parameters name or return value. In Function Signature, we calculate the cosine similarity between the name for the method source and the name for the method target. To reach this goal, We applied the following equation:

$$\begin{aligned} FS_{score(s,t)} = 0.25 * sim(rts_s, rts_s) + 0.25 * lfs(ips_s, ips_t) + \\ 0.5 * lfs(methodName_s, methodName_t) \end{aligned} \tag{4.2}$$

$lfs()$ calculate the most frequent sub-sequence between method name for the target and method name for the source.And, $sim()$ compute the token similarity between the return types.

4.2.4 TMAP

Another study that we compared our experiment with it is TMAP [10]. This study relies on the method description in order to do method mapping for the third-library documentations. It analysis the API documentation for source and target method by uses method description. In the following equation, TMAP collect five features that all related to the each method description:

$$TMAP_{score(s,t)} = \sum \varphi_1(\widehat{s},\widehat{t}) + \varphi_6(s,t) + \varphi_8(s,t) + \varphi_9(s,t) + \varphi_x(s,t) \tag{4.3}$$

Where $\varphi_x(s,t)$ is represents a score of how similar the source method class description cd_s to the target method class description cd_t. To extract this feature, first, we apply text reprocessing TR on the source/target method class description to clean sentences beside description. That will generate $\widehat{cd_s}$ from source method class description and $\widehat{cd_t}$ target method class description. then we just measure the cosine similarity between $\widehat{cd_s}$ and $\widehat{cd_t}$. For $\varphi_1(s,t)$, $\varphi_6(s,t)$, $\varphi_8(s,t)$, $\varphi_9(s,t)$ We already discussed in methodology how we extract these features.

4.3 Tuning

In this section we will elaborate the impact of tuning on the performance of the learner. Since, the two class boosted decision tree perform the best learner, we start our tuning as: *Maximum Number of leaves=2*, *Minimum leaf instances=4*, *Learning rate=0.06*, and *Number of trees=436*. And then we repeat the tuning process until we get small error that we cannot enhance anymore. Figure 4.1 shows the effect the tuning for the input of the decision tree and when we do tuning the accuracy increase from 87.5% to 91.1%. By applying tuning for our learner we were able to increase the accuracy for our learner.

Actual/Predicate	Yes	No
Yes	TP=48	FN=5
No	FP=7	TN=52

Table 4.1: Confusion Matrix for 30% of dataset

Table 4.1 shows confusion matrix, where we train model on 70% of dataset and we use 30% of dataset for testing. We can see the model we able to predicate 48 correct mapping as correct mapping (True Positive). The model we able to predicate 5 correct mapping as not correct mapping (False Negative). The model we able to predicate 7 not correct

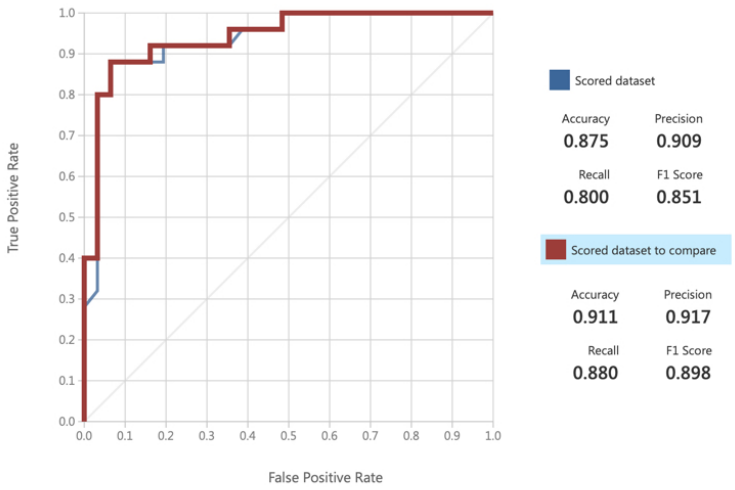


Figure 4.1: True positive and false positive ratio in ROC Curve with tuning and without tuning.

mapping as correct mapping (False Positive). The model we able to predicate 52 not correct mapping as not correct mapping (True Negative). That show the model is good in predicate the correct mapping as correct and not correct as not correct (True Positive and True Negative) and the data was distributed correctly between the two classes so there is no over-fitting issues. The example shows the confusion matrix RAPIM++ which is very similar to confusion matrix for RAPIM [11] and we will go more depth and do more comparison when we answering research question 2.

4.4 Results of RQ1.

We compute the accuracy across eight migration rule using five approaches. figure five 4.2 illustrate the result. Our result showed the accuracy diverse across the five rules from 70.3% to 95.8% percentages.Our approach RAPIM++ showed increase in accuracy by 2.9% comparing to state of art RAPIM which we extend our study from. This increase of accuracy approve that our approach was able to enhance the performance of the model. As, we discussed earlier our main contribution in this study is that to improve the recommenda-

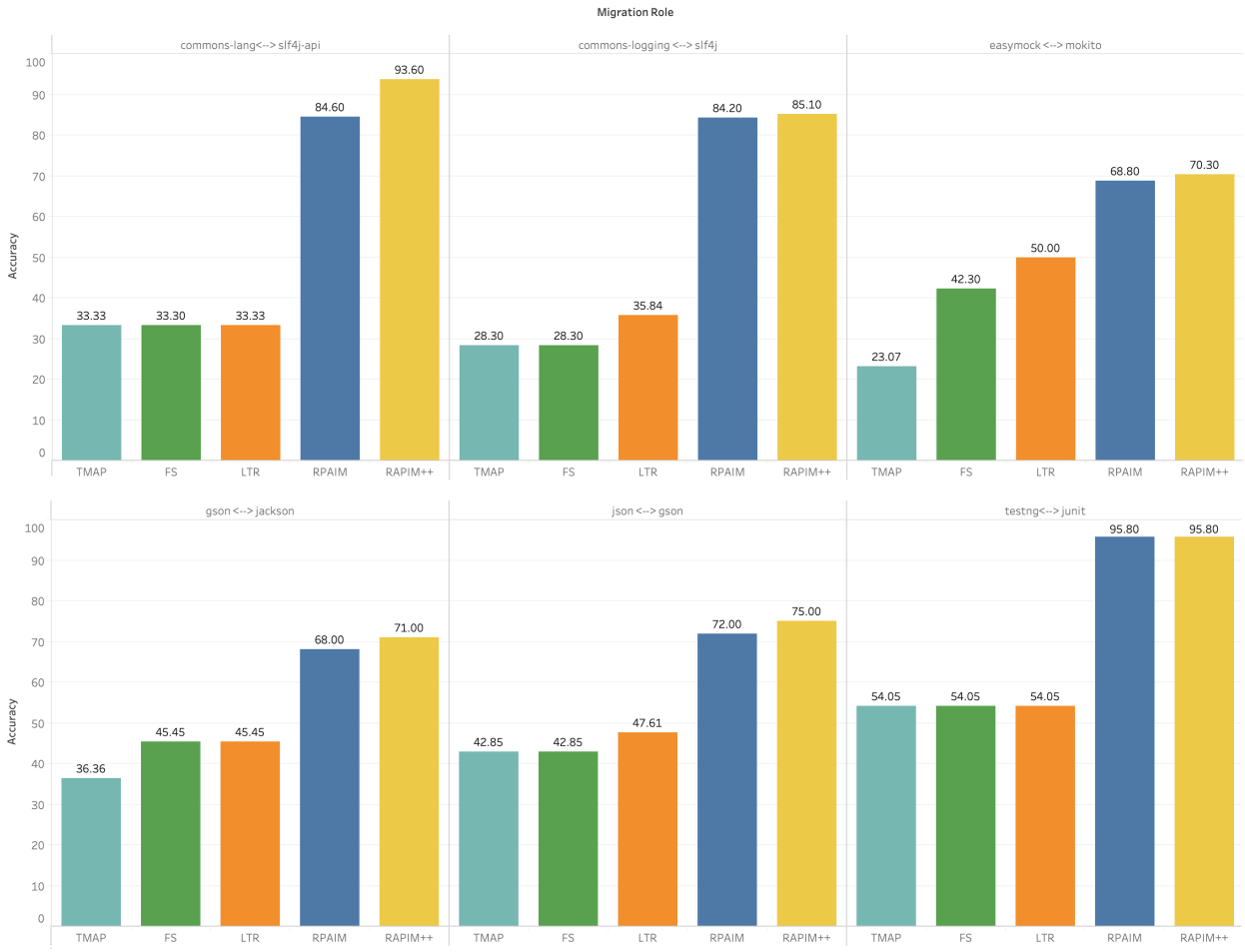


Figure 4.2: Comparison five different studies in case of accuracy, over 6 migrations rule .

tion of method mapping for libraries by using domain knowledge and source code knowledge. So lets take an example Figure 1.2 to see the impact of our approach to recommend the best mapping by switch method *put*(*String key, Map value*) by *addProperty*(*String key, String value*) while we utilize one of the 8 migration rules to migrate *json* library to *gson* library. Figure 4.4 illustrate source code implementation for *put*(*String key, Map value*) and *addProperty*(*String key, String value*). As shown in the figure *addProperty*(*String key, String value*) is not more than wrapper between two method, we used our parser that we discussed previously in order to spread the content of method *add*(*String property, JsonElement value*) as source code implementation for *addProperty*(*String key, String value*). Then we repeat the same process for *put*(*String key, Map value*) since it is just a wrapper

to for *put*(*String key, Object value*). Then We applied code2vec on both method source code implementation which creates that generate predicate topics for both methods. We also noticed that method signature for domain knowledge has different implementation *addProperty and put* while methods for source code implementation has same implementation *put,add,set*. We were able to indicate that source code knowledge allow more vision for the similarities between two methods. When we examine our result manually, we found some limitations. First: *Method Overloading*. this context means that when we have two methods in same name but different name of parameters, different type of parameters and different order of parameters. Second: *Polymorphic Methods.* this context means that when we have different types of subclass which has same name and same number of parameters of main class method. Third: when we have source method and target method that both different name, parameters and return type. Finally, the harder case was when the methods absence the documentations, that makes RAPIM, RAPIM++, LTR hard to recommend the correct mapping between the source and target method. By the end of this section, we can conclude that our approach was able to achieve the maximum accuracy by 84% , while the maximum accuracy that other approaches have achieved was 80.65%.

4.5 Results of RQ2.

Comparing to the state of the art RAPIM [11], RAPIM++ approved improving in terms of accuracy and Figure 4.3 showed the differences. We illustrate that, when we use 10% of the training data set we get 82% of accuracy,while there is a slightly different increasing the accuracy by 40% which give us 90% of accuracy. We can indicate that our approach is more stable and can recommend the correct method mapping though with sub-set of training data set. Also, our light weight web service will be more useful for users since even with sub-set training size will get a high accuracy.

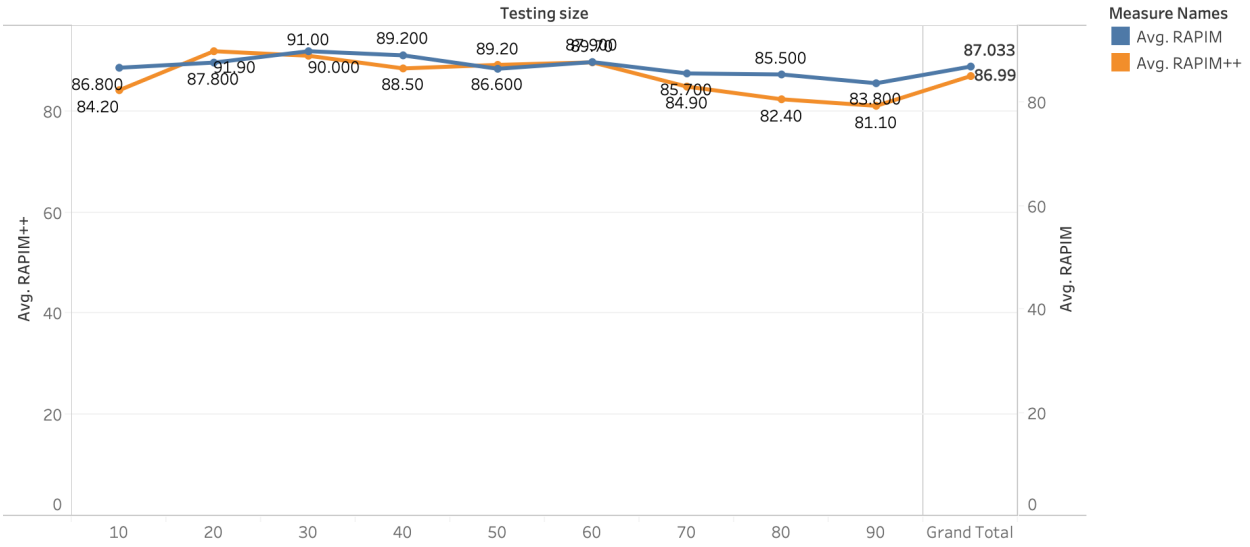
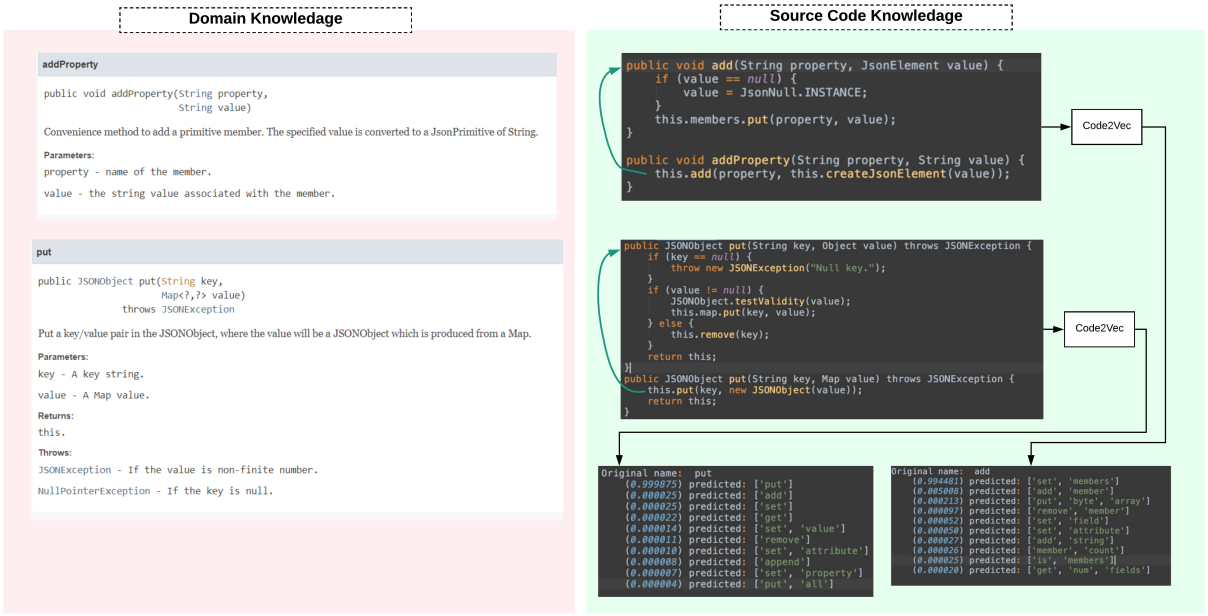


Figure 4.3: Effect of the size of the training data set .

4.6 Discussion and Challenges

For the feature extraction purpose, both case studies: RAPIM [11], and RAPIM++ depend heavily on the documentation for both domain knowledge and source code knowledge for the libraries. The documentations are very important to extract feature Q1,Q2 and Q3. However, we will able to extract all features from source code knowledge without the need for the documentations for libraries. we tested both approach RAPIM [11] and RAPIM++ for libraries which do not have documentations that were 21.6% of total numbers of libraries, when 78.4% of the total numbers of libraries include all the corresponding documentations.We build the model again and we excluded the three features Q1,Q2 and Q3. We named our new model RAPIM++– and we rerun the model again to calculate the accuracy and see the impact of the documentations on accuracy. Our finding was as fallow: the accuracy for RAPIM [11] 87.3%, while the accuracy for RAPIM++ was 89.99% on average. Our finding indicate that the high impact of documentations for libraries on the RAPIM [11] while RAPIM++ does not effect much when we exclude the documentations. This detecting approve that our model will be able to find the correct method mapping between unknown libraries which may do not have libraries.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Valid Mapping
RAPIM	0.6578812	0.5350229	-0.53730947	0.037593298	-0.09061304	-0.41274145	0.31988314	-0.36379427						No
RAPIM++	0.6578812	0.5350229	-0.53730947	0.037593298	-0.09061304	-0.41274145	0.31988314	-0.36379427	-0.21931201	-0.43375733	0.15219064	0.36600125	-0.735	Yes

Figure 4.4: Samples of RAPIM++ outperform RAPIM which illustrate method mappings between *json* and *gson*.

4.7 PCA Discussion

In this section, we discuss why using PCA was helpful in our experiment and how using PCA improves run-time and accuracy compared with not using PCA or using different models that previous studies used. We will do three different studies here:

4.7.1 PCA Feature Selection

Principal Component Analysis (PCA) is a machine learning concept which takes a huge numbers of features as an input and provide limited numbers of features as an output. The goal of the PCA is prevent any over fitting issues and reducing the effectiveness of noise. In our experiment, we used PCA to reduce the features of source code knowledge from 384 features for method source code for source method and 384 features for target method source code and present these huge numbers of features by reasonable numbers at same time keep the same contribution. We tried to present these features by 8 features like the same numbers which we extracted from the domain knowledge part. In fact, we figured out that representing these features by 8 features make our models not perform very well and run slower. Then we did tuning by trying to present it by 7 features then 6 features until we try one PCA.

In this section, we explain why and how we select the number of PCA features. As we know, we have 384*2 features from source code knowledge. It would be unfair to compare these features with only eight features from domain knowledge. So we need a way to make the number of features that we select from domain/source code knowledge useful and help in improving the experiment run-time. To do so, we need to run different experiments to find what is the best number of features that we have to pick from 384*4 source code knowledge features.

As shown in Table 4.2, We run ten experiments. we split our dataset into 70% training and 30% testing, and we run ten different experiments. For every experiment, we change the number of PCAs. We start with 384*2 code knowledge features as input and use PCA to generate one PCA then two PCAs, until ten PCAs as we can see in the Table 4.2, the runtime increase when we increase the number of features. Also, Accuracy, f-measure,

precision, and Recall change for different numbers of PCAs.

# Of PCAs	Accuracy	f-measure	precision	Recall	Runtime
1	91.1%	90.6%	90.6%	90.6%	50s
2	89.3%	88.7%	88.7%	88.7%	50s
3	88.4%	87.9%	87.0%	88.7%	50s
4	89.3%	88.9%	87.3%	90.6%	50s
5	89.3%	88.9%	87.3%	90.6%	50s
6	87.5%	87.0%	85.5%	88.7%	51s
7	90.2%	89.9%	87.5%	92.5%	51s
8	90.2%	89.7%	88.9%	90.6%	51s
9	86.6%	86.5%	82.2%	90.6%	51s
10	91.1%	91.1%	86.4%	96.4%	52s

Table 4.2: PCA Features impact

Depends on which quality matrix you want to look at to decide what is the best number of PCAs you want to pick. For our experiment, We pick five PCAs, and we think base on results in Table 4.2 any number of PCAs between 4-8 is good and works fine. It depends on the time you willing to wait for the experiment to run. We pick five PCA for two reasons: *First*, it takes less time to run, which is the 50s to run. *Second*, it gives high accuracy compared with other PCAs. However, we recommend making the number of PCAs features for source code knowledge the same number of features for domain knowledge, So you avoid any issues.

Even though ten and one PCAs give better results, however, we avoid them, because in this case, well can have over-fitting issues. In the case of one PCA, the domain knowledge will be dominated since we have one feature from source code knowledge and eight features from domain knowledge. While with ten PCAs, the source code knowledge is dominated, and the experiment takes more time to run.

4.7.2 PCA Vs CNN Model

In this section, We want to study which approach PCA or CNN Model [25] is better when we apply them on source code knowledge features. We want to measure them in term the impact on runtime and the model accuracy.

In order to answer this question, we did two experiments where we split our dataset into 70% training and 30% testing. In the first experiment, we use PCA approach, where we have eight features from domain knowledge and five PCAs features generated from 384*2 source code knowledge features. Then we feed these 13 features and one class to a boosted decision tree (BDT), and results are shown in Table 4.3. In the second experiment, we use CNN Model [25], where we have eight features from domain knowledge and one CNN score features that generated by feeding 384*2 features to CNN Model [25] plus one class. Then we feed these nine features plus output class to BDT.

Table 4.3 shows comparing the results of the two approaches. As we can see PCA outperform CNN model in two terms:

First, in term of prediction Accuracy (PCA=89.3% > CNN=87.5%), f-measure (PCA=88.9% > CNN =87.0%), precision (PCA=87.3% > CNN=85.5%), and Recall (PCA=90.6% > CNN=87.7%). We can see the PCA outperforms the CNN model in all these matrices.

Second, in terms of runtime, we can see PCA took 50 seconds to run while the CNN model took 5 minutes and 3 seconds, and that happens because the model needs to build Convolutional neural network (CNN) from thousands of interfaces for methods bodies. Base on these results, there is no need to use an external model. Instead, we use mathematics equations such as PCA here. PCA would be faster since we do not need to run the CNN model to build a neural network. Also, less costly since we do not need to collect datasets to train CNN model. So we decided to use PCA here. Also, with CNN Model, the domain knowledge will be dominated since we have one feature from source code knowledge and eight features from domain knowledge.

Approach	Accuracy	f-measure	precision	Recall	Runtime
PCA	89.3%	88.9%	87.3%	90.6%	50s
CNN Model	87.5%	87.0%	85.5%	87.7%	5m,3s
All Features	88.4%	88.1%	85.7%	90.6%	2m,6s

Table 4.3: Comparing results using with and without using PCA

Keep in mind these experiments run on the cloud machine, so the time is lease since the cloud has powerful hardware. If we run the experiment on a local machine, the run time will be much more. Furthermore, since we run in the cloud, we don't know how they

run the tasks or how they implement algorithms. They may get run tasks in parallel, so runtime may not be accurate.

4.7.3 Why PCA Is Helping?

In this section, We want to discuss how using PCA is helpful in terms of the impact on runtime and model accuracy and what will happen if we donot use PCA and just feed all 384*4 source code features to learner. To do so, We did two experiments where we split our dataset into 70% training and 30% testing. In the first experiment, we use PCA approach, where we have eight features from domain knowledge and five PCAs features generated from 384*2 source code knowledge features. Then we feed these 13 features and one class to a boosted decision tree (BDT), and results are shown in Table 4.3. In the second experiment, we did not use PCA. We feed all 384*2 source code knowledge features to the model, which means we have eight features from domain knowledge and 384*2 features from source code knowledge. Then we feed these 576 features plus output class to BDT. We call this experiment *All Features* (*AF*)

Table 4.3 shows comparing the results of the two approaches. As we can see PCA outperform all features approach in two terms:

First, in terms of prediction Accuracy (PCA=89.3% $>$ AF=88.4%), f-measure (PCA=88.9% $>$ AF=88.1%), precision (PCA=87.3% $>$ AF=85.7%), while only both have same Recall (PCA=90.6% = 90.6%). We can see the PCA outperforms the All Features in all these matrices.

Second, in terms of runtime, we can see PCA took 50 seconds to run while the All Features took 2 minutes and 6 seconds, which makes sense since when we have many features, the model needs more time to build and generate decision trees.

4.8 Threats to validity

Through next parts, we will clarify any possibilities that could impact our result and our experiment setup. These threats include:Internal Validity, External Validity, and Constrict Validity

The main threats to our methods is that the measurement that we used to calculate the features and running the experiment. Through out our experiment we fallowed the standard for the most popular frame works and libraries such as Microsoft AI [28]. Despite of this authenticated resources, it might be an error in measurement which we can consider it as an **Internal Validity**.

Besides the internal validity, there is another validity related to the scalability. Loss the documentation for the libraries which which will effect the performance of RAPIM++ since we extract the features from the documentation of the libraries.However, we relied on the most popular libraries that available on line along with their documentation.This **Construct Validity** may happens when use any library it may miss the method documentation for their libraries.

Finally, the threat that we have related to the performance of our result and we should mention that not all the methods of the libraries we were able to document it and this will have impact on our performance but this instances are limited to few numbers of methods. This threats known as **External Validity**.

Chapter 5

Future works

5.1 Abstract

In this chapter we conclude our main contribution in this study and how the novel approach RAPIM [11] helped us to extend our work from it. Furthermore, we included our possible future work in this chapter.

5.2 Conclusion

API migration always been a challenging task for developers as well as for companies. The difficulty for this task occurs because it is hard, error prone and costly. While previous studies were able to face these challenging by developing such approaches to recommend mapping between different libraries, non of them tried to focus on the recommendation at method level for different libraries.

The goal of this study is that indicate the challenges when automatically recommend mapping at method level between third part libraries, We have proposed RAPIM++ the extension of the state of the art RAPIM. We developed an automation approach to recommend mapping between two unknown libraries at method level. RAPIM++ uses the **Domain**

Knowledge and **Source Code Knowledge** for both the source library and the target to generate the important features in order to feed the model. Our model was able to predict the mapping between the methods of unknown libraries successfully. We conducted an empirical study to evaluate our result with the novel approach RAPIM where we extended our study. The comparison study conducted by comparing three algorithms from the RAPIM along with 8 poplar migration rules. The result showed that RAPIM++ was able to increase the accuracy by 2.9% where the accuracy was 86.97%. In addition we were able to serve the research and industry community by providing a website that include all the data set and all possible migrations between the APIs

5.3 Future works

Extend the data set that we have and make it more general is part from the future work for our study. In this case the developers will able to insert any APIs and our tool will be able to provide the possible mapping between the source and method libraries.

The study also plans to, not limit the java based libraries, when recommend the migration mapping between third party libraries. Including different programming languages from different open source projects, like C and python very important since the developers for different programming languages face challenging in migration mapping between libraries.

while we were able to build our model by using two boosted decision tree successfully, there are a numbers of machine learning algorithms that we can use in order to do comparisons between our model and other algorithms that designed to serve our problem. Conducting comparative study between different type of classifiers will be another part of our future study.

Our approach support One to One method mapping between source and target methods. We willing in our future work to include any type of mapping in order to provide comprehensive approach which able to detect the correct mapping between any type of migrations.

Finally, since the numbers of research papers are limited in this field, we wish to increase the publications in this specific field because the migration is very important part of software refactoring and the developers in industries faces serious problems in software maintenance, so it is important to investigate more and build more tools in order to provide the research and industry community with more comprehensive solution for all current limitations.

Bibliography

[1] M. Kim, D. Notkin, D. Grossman, Automatic inference of structural changes for matching across program versions, in: ICSE, Vol. 7, Citeseer, 2007, pp. 333–343.

[2] T. Schäfer, J. Jonas, M. Mezini, Mining framework usage changes from instantiation code, in: Proceedings of the 30th international conference on Software engineering, ACM, 2008, pp. 471–480.

[3] B. Dagenais, M. P. Robillard, Recommending adaptive changes for framework evolution, ACM Transactions on Software Engineering and Methodology (TOSEM) 20 (4) (2011) 19.

[4] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, M. Kim, Aura: a hybrid approach to identify framework evolution, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 1, IEEE, 2010, pp. 325–334.

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

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

[7] J. Härtel, H. Aksu, R. Lämmel, Classification of apis by hierarchical clustering, in: Proceedings of the 26th Conference on Program Comprehension, ACM, 2018, pp. 233–243.

[8] A. Gokhale, V. Ganapathy, Y. Padmanaban, Inferring likely mappings between apis, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 82–91.

[9] R. Pandita, R. P. Jetley, S. D. Sudarsan, L. Williams, Discovering likely mappings between apis using text mining, in: Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on, IEEE, 2015, pp. 231–240.

[10] R. Pandita, R. Jetley, S. Sudarsan, T. Menzies, L. Williams, Tmap: Discovering relevant api methods through text mining of api documentation, Journal of Software: Evolution and Process 29 (12) (2017).

[11] H. Alrubaye, M. W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, J. Mcgoff, Learning to recommend third-party library migration opportunities at the api level, Applied Soft Computing 90 (2020) 106140.

[12] H. Alrubaye, M. W. Mkaouer, Automating the detection of third-party java library migration at the function level, in: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2018, pp. 60–71.

[13] J. Shlens, A tutorial on principal component analysis, arXiv preprint arXiv:1404.1100 (2014).

[14] U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning distributed representations of code, Proceedings of the ACM on Programming Languages 3 (POPL) (2019) 1–29.

[15] M. Lamothe, W. Shang, T.-H. P. Chen, A3: Assisting android api migrations using code examples, IEEE Transactions on Software Engineering (2020).

[16] F. Thung, H. J. Kang, L. Jiang, D. Lo, Towards generating transformation rules without examples for android api replacement, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019, pp. 213–217.

[17] S. Xu, Z. Dong, N. Meng, Meditor: inference and application of api migration edits, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE, 2019, pp. 335–346.

[18] L. I. Martínez, C. Pereira, L. M. Favre, Migrating c/c++ software to mobile platforms in the adm context, International Journal of Interactive Multimedia and Artificial Intelligence 4 (2017).

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

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

[21] H. Alrubaye, M. W. Mkaouer, A. Ouni, On the use of information retrieval to automate the detection of third-party java library migration at the method level, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE, 2019, pp. 347–357.

[22] M. W. Hussein Alrubaye, Variability in library evolution, Software Engineering for Variability Intensive Systems: Foundations and Applications (2019) 295.

[23] H. Alrubaye, M. W. Mkaouer, A. Ouni, Migrationminer: An automated detection tool of third-party java library migration at the method level, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019, pp. 414–417.

[24] H. Alrubaye, D. Alshoaibi, E. Alomar, M. W. Mkaouer, A. Ouni, How does library migration impact software quality and comprehension? an empirical study, in: International Conference on Software and Software Reuse, Springer, 2020, pp. 245–260.

[25] H. A. T. Al-Rubaye, Towards the automation of migration and safety of third-party libraries (2020).

[26] T. Mitchell, B. Buchanan, G. DeJong, T. Dietterich, P. Rosenbloom, A. Waibel, Machine learning, Annual review of computer science 4 (1) (1990) 417–433.

[27] C. Teyton, J.-R. Falleri, X. Blanc, Automatic discovery of function mappings between similar libraries, in: In Reverse Engineering (WCRE), 2013 20th Working Conference on, IEEE, 2013, pp. 192–201.

[28] E. Loper, S. Bird, Nltk: the natural language toolkit, arXiv preprint cs/0205028 (2002).