LUISA FERNANDA HERNÁNDEZ RAMÍREZ

API RECOMMENDATION SYSTEM FOR SOFTWARE ENGINEERING

LAVRAS - MG

2016

LUISA FERNANDA HERNÁNDEZ RAMÍREZ

API RECOMMENDATION SYSTEM IN SOFTWARE ENGINEERING

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Banco de dados e Engenharia de Software, para a obtenção do título de Mestre.

Orientador:

Dr. Heitor Augustus Xavier Costa (DCC/UFLA)

LAVRAS - MG

2016

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Hernández Ramírez, Luisa Fernanda.
API recommendation system in Software Engineering / Luisa Fernanda Hernández Ramírez. – Lavras: UFLA, 2016.
223 p. : il.

Dissertação (mestrado acadêmico) – Universidade Federal de Lavras, 2016.
Orientador(a): Heitor Augustus Xavier Costa.
Bibliografia.

1. API recommendation. 2. Collaborative Filtering. 3. Frequent Itemset Mining. 4. Evaluation metrics. 5. Recommendation system. I. Universidade Federal de Lavras. II. Título.

LUISA FERNANDA HERNÁNDEZ RAMÍREZ

API RECOMMENDATION SYSTEM IN SOFTWARE ENGINEERING

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Banco de dados e Engenharia de Software, para a obtenção do título de Mestre.

APROVADA em 21 de Março de 2016

Dr. Marco Túlio de Oliveira Valente	UFMG/DCC
Dr. André Pimenta Freire	UFLA/DCC
Dr. Paulo Afonso Parreira Júnior	UFLA/DCC

Dr. Heitor Augustus Xavier Costa
Orientador.

LAVRAS - MG

2016

A meu pais, Luis Eduardo Hernández e Beatriz Ramírez pelo legado de amor, honestidade e responsabilidade, cujo incentivo é muito importante para seguir adiante. Além disso, pelo seu amor, sua compreensão e incentivo de responsabilidade em minha vida pessoal e laboral. Amo muito vocês!

A minhas irmãs, Ximena Hernández, Alexandra Chavarro e Maritza Chavarro, que são um apoio vital em minha vida, e sempre são um exemplo de superação e amor; Amo muito vocês!

A meus sobrinhos e sobrinhas que são um motor para seguir adiante e ser um exemplo de pessoa e professional. Amo muito vocês!

A minha família em Bogotá porque tem sido um apoio importante na minha formação como pessoa e professional. Amo muito vocês!

À Jordano Salamanca por seu amor, pela compreensão e paciência, por estar comigo nos momentos tristes e de felicidade, e especialmente pelo apoio incondicional. Te amo muito!

A minhas melhores amigas por ser parte de minha vida e porque sei que ainda estando longe sua amizade é incondicional. Amo muito vocês!

DEDICO

AGRADECIMENTOS

À Universidade Federal de Lavras e ao Departamento de Ciência da Computação, pela oportunidade para a realização do mestrado;

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela concessão da bolsa de estudos.

Ao Professor Heitor Costa, pela orientação, apoio e confiança para a realização do mestrado. Sinto-me muito privilegiado, em todos os aspectos, por ter tido a oportunidade de tê-lo como orientador;

Ao professor Marco Tulio Valente pela colaboração, as sugestões e as correções, que contribuíram muito para o desenvolvimento desse trabalho, além de sua participação na banca examinadora;

Ao André Hora pela participação, colaboração, e sugestões que contribuíram muito para o desenvolvimento deste trabalho.

Aos professores do DCC da UFLA e colegas pós-graduandos pelos ensinamentos transmitidos e conhecimentos compartilhados.

A todos aqueles que, de alguma forma, contribuíram para a realização desse trabalho.

Sinceramente agradeço.

ABSTRACT

Software development depends on Application Programming Interfaces (APIs) to achieve their goals. However, choosing the right APIs remains as a difficult task for Software Engineers. In software engineering, recommendation systems are emerging to support Software Engineers in their decision-making tasks. Therefore, in this work, we proposed a methodology that considers software categories and recommends APIs to Software Engineers with software in initial (not using APIs) or advanced (using some APIs) stage of software development. We used collaborative filtering technique along with frequent Itemset mining technique for generating the corresponding large and top-*N* lists of APIs recommended. In the top-*N* lists, the goal was to find a few specific APIs that are supposed to be most useful to Software Engineers. In order to automate the methodology proposed, we developed a plug-in for the Eclipse IDE. In addition, we tested the methodology considering categories from the SourceForge open source repository. For every category, we evaluated large and top-*N* lists performance based on two classification accuracy metrics (precision and recall) and one efficacy metric (recall rate), obtaining promising outcomes. Thus, the results of evaluation metrics showed that our methodology could make useful API recommendations for Software Engineers with software that used a small number of APIs or did not use any API. Besides, our methodology was able to put relevant APIs even in high-ranking positions, even in small top-*N* lists of APIs recommended.

Keywords: API recommendation; Collaborative Filtering; Frequent Itemset Mining; Evaluation metrics; Recommendation system.

RESUMO

Desenvolvimento de software depende Interfaces de Programação de Aplicações (APIs - *Aplication Programming Interfaces*) para atingir seus objetivos. No entanto, a escolha correta dessas APIs permanece como uma tarefa difícil para Engenheiros de Software. Na Engenharia de Software, os sistemas de recomendação estão surgindo para apoiar aos Engenheiros de Software nas tarefas de tomadas de decisões. Portanto, neste trabalho foi proposta uma metodologia que considera categorias de sistemas de software e recomenda APIs para engenheiros de Software com sistemas na etapa inicial (sistemas que não usam APIs) ou na etapa avançada (sistemas que usam algumas APIs) de desenvolvimento de software. As técnicas de Filtragem Colaborativa (CF - *Collaborative Filtering*) e de Mineração de Itens mais Frequentes (FIS - *Frequent Itemset mining*) foram utilizadas para gerar as listas longas e curtas (top-*N*) das APIs recomendadas. Nas listas curtas (top-*N*), o objetivo foi encontrar um número *N* de APIs especificas que possivelmente seriam mais uteis para os Engenheiros de Software. Para automatizar essa metodologia proposta foi desenvolvido um *plug-in* para a plataforma Eclipse. Além disso, a metodologia foi testada considerando as categorias do repositório de código aberto SourceForge. Para cada categoria, foi avaliado o desempenho das listas longas e listas curtas (top-*N*) usando duas métricas de acurácia (*precision* e *recall*) e uma métrica de eficácia (*recall rate*), obtendo resultados prometedores. Esses resultados das métricas de avaliação mostraram que a metodologia conseguiu fazer recomendações de APIs uteis para os Engenheiros de Software com sistemas de software que usavam algumas APIs o que não utilizavam APIs. Do mesmo modo, a metodologia foi capaz de colocar APIs relevantes em posições altas das listas, inclusive nas listas curtas de APIs recomendadas.

Palavras-chave: Recomendação de APIs; Filtragem colaborativo; Mineração de Itens mais Frequentes; Métricas de avaliação; Sistema de recomendação.

FIGURE SUMMARY

TABLE SUMMARY

SUMMARY

1 INTRODUCTION

Despite steady advancement in the state of the art, software development remains a challenging and knowledge-intensive activity (ROBILLARD et al., 2014). Thus, software development is not an easy activity. Proof of that is the number of different components, which affect different aspects of the development process (CANÓS; LETELIER; PENADÉS, 2003). Among those components, there are software libraries known as Application Programming Interfaces (APIs). The usage of APIs is becoming a larger part of programming; there are more APIs than ever and their size is growing (STYLOS; MYERS, 2007). Possible causes of this growth are the benefits provided by APIs. For instance, they provide reusable functionalities and increase development productivity and software quality (SUN; KHOO; ZHANG, 2011). Furthermore, software reuse is often achieved using frameworks and libraries, whose functionality is exported through APIs (ROBILLARD et al., 2013).

Software Engineers have difficulty to select effectively the APIs to reuse during software development and selecting APIs appropriately is important because several studies have addressed the issue of APIs learning and usage in order to support software engineers (MILEVA et al., 2009; RUPAKHETI; HOU, 2011; MONTANDON et al., 2013). Then, effectively using APIs remains a challenge for Software Engineers because they may not become aware of these APIs as they are released and developers may thus be led to “re-implement the wheel” (THUNG et al., 2013). This difficulty can be resolved by finding and hiring API experts (TEYTON et al., 2013a), but it requires high investment (e.g., external human resources, availability, and money) that could not be included in the development budget. Thus, choosing the right APIs remains a difficulty for Software Engineers and some other strategies could be used like considering recommendation techniques and/or data mining techniques.

1.1 Motivation

A software library is a collection of classes that aim to facilitate software development. A library contains code and data, which provide services and can be exported via APIs (SUN; KHOO; ZHANG, 2011). Hence, an API is the interface to a reusable software entity used by multiple clients outside the developing organization and which can be distributed separately from the environment code (ROBILLARD et al., 2013).

Software development is inseparable from the use of APIs (DUALA-EKOKO; ROBILLARD, 2012) due to advantages of reusing them. For instance, APIs provide a cost-effective way to build software with enhancements in (SUN; KHOO; ZHANG, 2011): i) productivity of Software Engineers by providing a variety of desired functionalities; ii) software quality, as libraries are usually well-tested and fairly robust thanks to their massive and diverse user base.

API reuse saves development time because it prevents Software Engineers to redevelop existing services and features (TEYTON; FALLERI; BLANC, 2013). Nevertheless, the increasing size of APIs and the increase in number of APIs available imply in Software Engineers having to frequently learn how to use unfamiliar APIs (DUALA-EKOKO; ROBILLARD, 2012). Hence, it is difficult for Software Engineers to effectively or correctly reuse these APIs during their activities (ACHARYA et al., 2007).

Consequently, Recommendation Systems in Software Engineering (RSSEs) could be used because they provide reduction in effort, increment of productivity in software engineering activities, and support in users decision-making (ROBILLARD et al., 2014). There are studies whose aim is to support Software Engineers with low experience on APIs (Chapter 10). A study served as base for this work proposed a hybrid approach to recommend APIs, combining association rule mining and nearest neighbor based on collaborative filtering (THUNG; LO; LAWALL, 2013). In spite of the promising results presented in

that study, we observed some limitation that motivated the present work. Some of those limitations were: i) Recommendations were based and made just for maven software, minimizing the number of software data sample; and ii) The recommendation approach did not consider software in initial stage of development (i.e., in design or with deployed code, but without usage of APIs), instead, target software of recommendation must use some APIs.

Thus, the main motivation of this project lies in the fact that, in several studies found in the literature, there are neither recommendation methodologies or API recommendation systems that consider application categories nor recommend to Software Engineers in initial stage of software development (in design or with deployed code, but without usage of APIs). Then, we would be able to contribute for software engineering by supporting Software Engineers in their decision-making about which APIs to reuse during the software development process when they already consider software categories.

1.2 General objective

The main objective of this study was to perform an empirical analysis of software concerning their application categories, usage, and reusability of Application Programming Interfaces (APIs). Therefore, we established a methodology for identifying similar software and most commonly used APIs for a sample of software categories. Furthermore, to automate such identification, we developed an API recommendation system providing support for Software Engineers in their software development and/or maintenance activities and serving as a guide on making-decisions about API reuse.

For achieving general objective, we established the following objectives:

a) To plan a research method;
b) To create a methodology that considers software categories and recommends APIs for Software Engineers with software in initial stage

of development (which do not use APIs) or in advanced stage of development (which may use some APIs);

c) To get a dataset with as large number as possible of Java software;
d) To classify and filter software dataset into application categories;
e) To automate the methodology proposed by developing a Plug-in for the Eclipse IDE in order to support Software Engineers' activities when developing or maintaining software in initial stage of development (which do not use APIs) or in advanced stage of development (which may use some APIs);
f) To perform evaluation of the API recommendation system;
g) To analyze and discuss evaluation results.

1.3 Structure

The remainder of this work is organized as follows. **Chapter 2** presents the research method, its phases and activities. **Chapter 3** introduces recommendation systems, their importance in the Software Engineering area, their advantages and disadvantages, and the recommendation technique selected for this work. Besides, it shows evaluation methods and metrics commonly used for evaluating recommendation systems. **Chapter 4** presents the proposed methodology to recommend APIs for Software Engineers and a strategy to evaluate it. **Chapter 5** exposes the development of the proposed methodology. **Chapter 6** shows results of evaluating the recommendation methodology. **Chapter 7** discusses results and their implications. **Chapter 8** discusses threats to validity. **Chapter 9** presents related works. **Chapter 10** presents conclusions, threats to validity, and future work.

2 RESEARCH METHOD

We elaborated a research method in order to define phases and activities for creating a methodology to support Software Engineers by using API recommendations. The phases and activities made it possible to establish the theoretical grounding, the data collection and manipulation, the recommendation strategy, and the computational support with its evaluation.

We presented the four phases of the research method (Figure 1). Each phase is comprised of activities. We explained each phase and activity as follows:

Phase 1: Bibliographic research: Establishment of activities for collecting and understanding relevant and useful material for defining a methodology to recommend APIs for Software Engineers with software in initial or advanced stage of development.

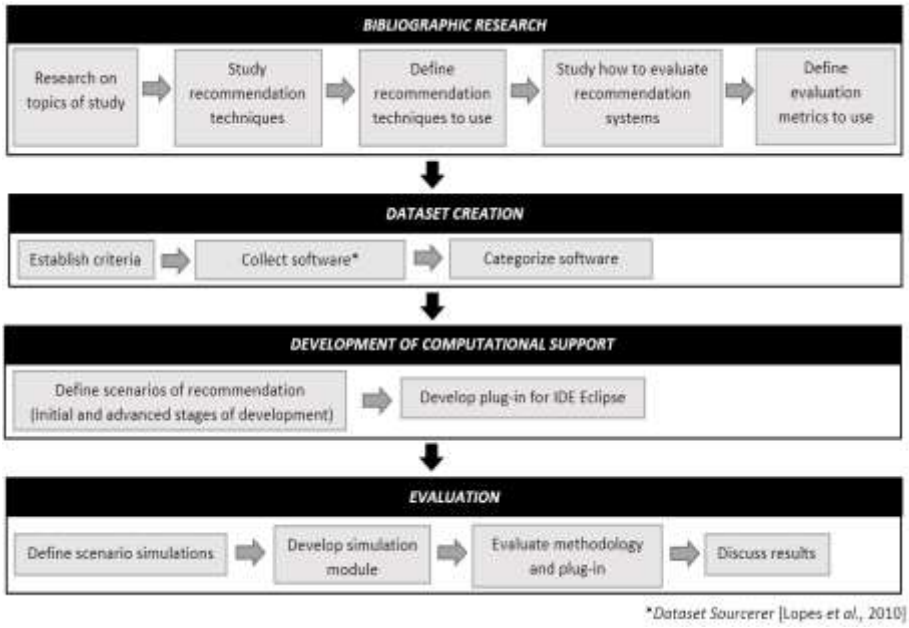

Figure 1 Research method.

Activity 1.1. Research on topics of study. Collection, comprehension and selection of relevant topics. Those topics were: information retrieval, recommendation systems, recommendation techniques, evaluation of recommendation systems, Application Programming Interfaces (APIs), open source repositories, software categories, and development of plug-ins for IDE-Eclipse;

Activity 1.2. Study recommendation techniques. Collection and comprehension of recommendation techniques used in different contexts of recommendation systems;

Activity 1.3. Define recommendation techniques to use. Selection of appropriate recommendation techniques based on context and criteria of API recommendation;

Activity 1.4. Study how to evaluate recommendation systems. Collection and comprehension on techniques and metrics for evaluating of recommendation systems;

Activity 1.5. Define evaluation metrics to use. Selection of techniques and metrics for evaluating the API recommendation methodology.

Phase 2: Dataset creation: Definition of criteria for collecting and classifying software in order to obtain real baseline software.

Activity 2.1. Establish criteria. Establishment of criteria for selecting relevant software, e.g. repository, programming language, platform, number of APIs, etc;

Activity 2.2. Collect software. Collection and filtering of software that the meet criteria established;

Activity 2.3. Categorize software. Classification of software collected into categories.

Phase 3: Development of computational support. Definition of a strategy for automating the methodology of recommendation by using a computational support, i.e., a plug-in for IDE Eclipse.

Activity 3.1. Define scenarios of recommendation (Initial and advanced stages of development). Definition of two scenarios for supporting Software Engineers by using API recommendations. In this activity, we established criteria and recommendation methodology for initial and advanced stages of software development;

Activity 3.2. Develop plug-in for the Eclipse IDE. Automation of defined strategy for recommending by developing a plug-in for the IDE Eclipse.

Phase 4: Evaluation. Automation of the evaluation of recommendation methodology proposed.

Activity 4.1. Define scenario simulations. Establishment of criteria, context and strategy to simulate real users in every scenario defined;

Activity 4.2. Develop simulation module. Development of a module into the plug-in for simulating real users as defined in Activity 4.1;

Activity 4.3. Evaluate methodology and plug-in. Execution of statistical tests in order to evaluate results generated by the simulation module, without real users' interaction;

Activity 4.4. Discuss results. Discussion and conclusion of results, limitations, and future work.

3 BACKGROUND

The area of Recommendation Systems (RSs) began in the nineties. The first recommendation system was called Tapestry (GOLDBERG et al., 1992). Then, the first papers emerged on a recommendation technique known as Collaborative Filtering (CF), in which the recommendation problem with ratings was formalized and intensely studied (SOUZA, 2011). New recommendation techniques have been developed and they are interesting for both, industry and academia (MARTINS, 2013). As consequence, many strategies for evaluating recommendation systems surged.

The remainder of this chapter is organized as follows. Section 3.1 introduces recommendation systems. Section 3.2 introduces the frequent itemset mining technique that is usually used along with recommendation techniques. Section 3.3 presents methods for evaluation recommendation systems.

3.1 Recommendation systems

Recommendation systems are software applications that aim to support users in their decision-making. They filter and recommend items of interest to users based on preferences they have expressed, either explicitly or implicitly (ROBILLARD; WALKER; ZIMMERMANN, 2010). With that filter, RSs also aim to solve the problem of data overload and then, they allow to discover new content faster and more efficiently and to support discovery rather than search (ANAND; BHARADWAJ, 2011; NÚNEZ-VALDÉZ et al., 2012).

There are three major processes in recommendation (BIGDELI; BAHMANI, 2008): i) object data collections and representations. It consists of getting the items to be analyzed; ii) similarity decisions. It involves calculation of

distance/similarity between data; and iii) recommendation computations. It introduces and filters recommendation of relevant items.

The remainder of this chapter is organized as follows. Section 3.1 introduces the importance of recommendation systems in software engineering. Section 3.2 argues advantages and disadvantages of RSs. Section 3.3 exposes Collaborative Filtering technique. Section 3.4 summarizes the Frequent Itemset mining technique.

3.1.1 Importance of recommendation systems in Software Engineering

A Recommendation System in Software Engineering (RSSE) is a software application that provides information items considered as valuable for a software engineering task in a given context. RSSEs are emerging to assist Software Engineers in various activities, from reusing code to writing effective bug reports (ROBILLARD; WALKER; ZIMMERMANN, 2010). Such activities are increasing because Software Engineers are continually introduced to new technologies, components, and ideas (ROBILLARD et al., 2014).

For RSSEs, no reference architecture has emerged to date (ROBILLARD et al., 2014). Nevertheless, in RSSEs, three design dimensions are commonly considered (Table 1) (ROBILLARD; WALKER; ZIMMERMANN, 2010):

a) **Nature of the context**. It is the RSSE input and consequently is a core concept. That input can be explicit (e.g., user-interface interactions such as entering texts, selecting elements in the code, etc.), implicit (e.g., track and react to developer actions), or a hybrid of these strategies;
b) **Recommendation engine**. It consists in on context data to make recommendations and on the ranking mechanisms;
c) **Output mode**. It refers to the mode used for presenting recommendations to the users. Some modes are pull mode, push mode, batch mode, and inline mode. Pull mode produces recommendations after a developer's

explicit requests, which can be as simple as a single click in an IDE. Push mode delivers results continuously. In batch mode, developer wants a complete set of recommendations about a task and is therefore willing to go to a separate IDE view. Finally, in inline mode, annotations are made atop artifacts that the developer is otherwise perusing.

Traditional RSs provide a variety of functions. In addition, to assist users in a number of ways, these functions also include a number of benefits to other stakeholders, including commercial organizations (e.g. increasing the number of items sold, selling more diverse items, and increasing customer loyalty). On the other hand, in the case of RSSEs, most research studies have focused on the support directly provided to Software Engineers (ROBILLARD et al., 2014).

Table 1 RSSE design dimensions

<table>
<tr><th>Nature of the context</th><th>Recommendation engine</th><th>Output mode</th></tr>
<tr><td rowspan="3">Input:
explicit | implicit | hybrid</td><td>Data:
source | change | bug reports | mailing lists | interaction history | peers' actions</td><td>Mode:
push | pull</td></tr>
<tr><td>Ranking:
yes | no</td><td>Presentation:
batch | inline</td></tr>
<tr><td colspan="2">Explanations:
from none to detailed</td></tr>
<tr><td colspan="3">User feedback:
none | locally adjustable | individually adaptive | globally adaptive</td></tr>
</table>

Source: Robillard, Walker and Zimmermann (2010)

RSs provide recommendations for items of potential interest for a user, answering questions such as (ROBILLARD et al., 2014):

a) which book to buy?;
b) which website to visit next?; and
c) which financial service to choose?

On the other hand, in Software Engineering scenarios, questions that can be answered with support of RSs are (ROBILLARD et al., 2014):

d) which software changes probably introduce a bug?;
e) which requirements to implement in the next software release?;
f) which stakeholders should participate in the upcoming software project?;
g) which method calls might be useful in the current development context?;
h) which software components (or APIs) to reuse?.

3.1.2 Advantages and disadvantages

Among the advantages of using RSs and RSSEs, we cited the reduction in effort, the increment of productivity in software engineering activities, and the support in users decision-making (ROBILLARD et al., 2014). In addition, they allow discovering new content faster and more efficiently (NÚNEZ-VALDÉZ et al., 2012).

Regarding disadvantages, the main one of RSs is the need of enough data from users and/or enough users. That is known as the cold-start problem. This problem has two variants (SON, 2014):

a) **New item cold-start problem**. It occurs when there is a new item in the RS and, because it is a new product, there are not user ratings/usage and new items can be ranked at the bottom of the recommended items;
b) **New user cold-start problem**. It occurs when there is a new user, making difficult to give the prediction to a specific item for new user. Technique recommendations, such as collaborative filtering and content-based filtering, require an historic rating of this user to calculate similarities for defining the neighborhood. For this reason, new user cold-start problem can negatively affect recommendation performance due to the inability of the system to produce meaningful recommendations.

3.1.3 Collaborative filtering recommendation technique

There are six different classes of recommendation techniques (RICCI et al., 2011): i) Content-based; ii) Collaborative filtering; iii) Demographic; iv) Knowledge-based; v) Community-based; and vi) Hybrid recommendation systems. The choice of recommendation techniques depends on context, users, and items to be recommended. For example, items can vary from a book, a film, social networks friends to a software component. In each example, an appropriate technique must be used.

Among different techniques used by recommendation systems, Collaborative Filtering (CF) is the most widely used and effective recommendation technique (ANAND; BHARADWAJ, 2011) and CF has been used in many real systems, including environmental sensing, financial services, electronic commerce, web applications, etc. (THUNG; LO; LAWALL, 2013).

CF is based on the assumption that users who have agreed to something in the past tend to agree in the future (ANAND; BHARADWAJ, 2011). With CF, when there is enough information stored on the system (e.g. users ratings about set of elements), it is possible to make recommendations to each user based on information provided by those users considered to have the most in common with them (BOBADILLA et al., 2013). For that reason, it is possible to assert that CF is based on opinions and preferences of other users who are considered similar to the target software of recommendation.

The GroupLens project of the University of Minnesota is one of the pioneering researcher groups in the development of CF. That group elaborated basis algorithms for recommendation systems (GALÁN, 2007). There are two general methods of Collaborative Filtering algorithms, therefore, CF can be classified in (BREESE; HECKERMAN; KADIE, 1998):

a) **Memory-based**: Operates over the entire user database to make predictions (BREESE; HECKERMAN; KADIE, 1998). Memory-based

are heuristics-based algorithms, which use the entire rating history to arrive at predictions (ANAND; BHARADWAJ, 2011). Briefly, memory-based method calculates similarities among users and selects the most similar users as the neighbors of the active user (Nearest Neighbor - *NN*). Next, it gives recommendations according to the neighbors. Therefore, this method can give considerable recommended accuracy, but its computation time grows rapidly with the increase of users and (LIU et al., 2014). Memory-based methods usually use similarity metrics to obtain the distance between two users, or two items, based on each of their ratios (BOBADILLA et al., 2013). Nearest-Neighbor (*NN*) or user-based collaborative filtering is more popular and widely used in practice [Sarwar *et al*., 2001];

b) **Model-based**: In contrast to memory-based, model-based method uses user database to estimate or to learn a model used for predictions (BREESE; HECKERMAN; KADIE, 1998). Model-based use RS information to create a model that generates the recommendations. Algorithms in this category take a probabilistic approach and envision the Collaborative Filtering process as computing the expected value of a user prediction, given his/her ratings on other items [Sarwar *et al*., 2001]. Then, we consider a model-based method if new information from any user outdates the model. Among the most widely used models are Bayesian classifiers, neural networks, fuzzy systems, genetic algorithms, latent features, and matrix factorization (BOBADILLA et al., 2013).

3.1.3.1 Overview of the collaborative filtering process

Collaborative Filtering (CF) relies on two different types of background data (ROBILLARD et al., 2014): i) a set of users; and ii) a set of items. Thus, in a typical CF scenario, the background is:

a) Current user U_a
b) List of users $U = \{U_1, U_2, U_3 \ldots U_x\}$, where *x* is number of users;
c) List of items $I = \{I_1, I_2, I_3 \ldots I_n\}$, where *n* is number of items.

The relationship between users and items is primarily expressed in terms of ratings. Those ratings are provided by users and exploited in future recommendation sessions for predicting the rating that a user (U_a) would provide for a specific item. If user U_a interacts with a CF recommendation system, the first step of the RS is to identify the nearest neighbors (similar users compared to U_a) and to extrapolate from the ratings of the similar users the rating of U_a (ROBILLARD et al., 2014).

With background data exposed above, Figure 2 shows dataflow of a recommendation system based on CF process, where U_a is rating items and receiving recommendations of items based on ratings of users with similar rating behavior. Those users with similar behaviors are known as nearest neighbors (NN) (ROBILLARD et al., 2014). In that dataflow, U_a rated some items and the CF technique let to identify similar users based on the background data, i.e., based on ratings given by model users to items. Afterwards, CF used the rating information from similar users to rank and present recommendations to U_a. In addition, we must consider that computation of the NN set of users or items is the most important step of recommendation systems and the key is the calculation of similarity (DAPENG; QIANHUI; JINGMIN, 2009).

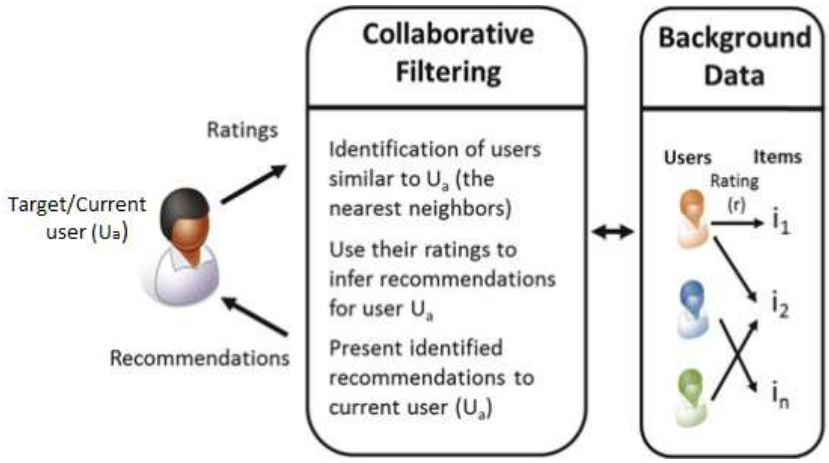

Figure 2 Collaborative filtering (CF) dataflow
Source: Robillard et al. (2014)

3.1.3.2 Overview of collaborative filtering algorithms

Collaborative Filtering (CF) algorithms vary depending on the approach in which the CF technique is based. There are two basic approaches to Collaborative Filtering, where both variants are predicting to which extent the active user would be interested in items which have not been rated by her/him up to now (ROBILLARD et al., 2014). Those two approaches are [Konstan *et al*., 1997] [Sarwar *et al*., 2001]:

a) **User-based algorithms**. User-based CF is the most successful approach for building recommendation systems to date and it is extensively used in commercial area. Let *R* be an *x*×*n* user-item matrix containing historical ratings information of *x* users $U = \{U_1, U_2, U_3 \dots U_x\}$ on *n* items $I = \{I_1, I_2, I_3 \dots I_n\}$. Let *P* be the set of items that have already been rated by the user U_a for which is wanted to compute the top-*N* recommendations. Thus, User-based CF *recommendation* systems compute the top-*N* recommended items for that user as follows. First, the most similar users (Nearest Neighbors - NN) to the current user U_a in the database are

identified. This is often done by modeling the users and items with the vector-space model, widely used for information retrieval. In that model, each user from the list of users and U_a are treated as a vector in the *n*-dimensional item space and similarity between U_a and the existing users is measured by computing similarity between among vectors (KARYPIS, 2001). Once that set of the NN users have been discovered, the group and their frequency aggregate their corresponding rows in R to identify the set C of items purchased. Using this set, user-based CF techniques recommend the top-*N* most frequent items in *C* that are not already in *P* (i.e., U_a has not already purchased) (KARYPIS, 2001). Consequently, top-*N* recommendation is organized by their frequency value in order to prioritize by useful the items to recommend.

b) **Item-based algorithms**. These approaches analyze the user-item matrix to identify relations between different items and they use these relations to compute the list of top-*N* recommendations. The key motivation of these schemes is that a user will more likely purchase items that are similar or related to the items that he/she has already purchased. Since, these schemes do not need to identify the neighborhood of similar users when a recommendation is requested, they lead to much faster recommendation engines (KARYPIS, 2001). Thus, Item-based CF recommendation systems use item-to-item similarity to compute the relations among the items and they compute the top-*N* recommended items for a target user. An Item-based CF algorithm behaves as follows. First, for each item *I*, i.e., $\{I_1, I_2, I_3 \dots I_n\}$, the *k* most similar items $\{I_1, I_2, \dots, I_k\}$ are computed and their corresponding similarities $\{sI_1, sI_2, \dots, sI_k\}$ are recorded. After, for each user *U*, i.e., $\{U_1, U_2, U_3 \dots U_x\}$, that has ranked a set (i.e., basket, movies, software components) *P* of items *I*, this ranking information is used to compute the top-*N* recommended items.

Therefore, the set C of candidate recommended items is identified by taking the union of the k most similar items for each item $I \in P$ and removing from the union any items that are already in P. Then, similarity for each item $c \in C$ is computed to the set P as the sum of the similarities between all the items $I \in P$ and c, using only the k most similar items of I. Finally, items in C are sorted in non-increasing order with respect to that similarity and first N items are selected as top-N recommended (KARYPIS, 2001).

3.1.3.3 Overview of similarity measures

The computation of the Nearest Neighbor (NN) set of objects (in our case, those objects are users or items) is the most important step of the personalized recommendation system and the key to this step is the calculation of similarity (DAPENG; QIANHUI; JINGMIN, 2009). To calculate that similarity among objects and to map them into a single numeric value, we can use similarity and distance metrics (HUANG, 2008). Broadly, similarity is the inverse of distance and the greater the distance between two objects, less similar they are (SAEED et al., 2003).

Traditional similarity metrics, as Cosine, Jaccard, and Pearson Correlation can be used for computing that similarity (DAPENG; QIANHUI; JINGMIN, 2009; BOBADILLA et al., 2011). We showed some of them below, assuming that letters A and B in the equations represent objects, i.e., users or items:

a) **Jaccard similarity**. Jaccard (Eq. 1) is a metric often used for comparing similarity, dissimilarity, and distance of the data set. Measuring the Jaccard similarity between two data sets is the result of division between the number of features that are common to all divided by the number of properties (NIWATTANAKUL et al., 2013). Jaccard is a similarity

metric and ranges [0, 1], where 0 means completely different and 1 means total similarity (HUANG, 2008).

$$SIM_{Jaccard}\ (A, B) = |A \cap B| \ / \ |A \ U \ B| \qquad \text{(Eq. 1)}$$

b) **Cosine similarity**. Cosine similarity (Eq. 2) is one of the most popular similarity metrics applied to text documents, such as numerous information retrieval applications and clustering. As result, the cosine similarity is non-negative and ranges [0, 1] where 0 is nothing similar and 1 is total similarity (HUANG, 2008).

$$SIM_{Cosine}\ (A,B) = |A \times B| \ / \ |A \cdot B| \qquad \text{(Eq. 2)}$$

c) **Pearson similarity**. Pearson's correlation (Eq. 3) is a metric of the extent to which two vectors are related. However, unlike the other metrics, it ranges from [-1, +1], where -1 means completely different and +1 means total similarity (HUANG, 2008). There are different forms of the Pearson correlation coefficient formula. In the standard formula (Eq. 3), σ represents the standard deviation between the data set *A* and *B* and Cov(*A*, *B*) represents the covariance (PARAMBATH, 2013).

$$SIM_{Pearson}\ (A,B) = Cov(A,B) \ / \ \sigma A \sigma B \qquad \text{(Eq. 3)}$$

3.2 Frequent itemset mining

Data mining is the process of analyzing the data from different perspectives and going over the useful information used to increase revenue, cuts costs, or both. Precisely, data mining is the process of finding correlation or patterns among dozens of fields in large relational database (PARMAR; SUTARIA; JOSHI, 2014). In data mining, these problems are solved by using some techniques known as Frequent Itemset mining (FIS) and Association Rules mining (AR) (AGRAWAL; IMIELINSKI; SWAMI, 1993). Those techniques are

commonly used along with recommendation techniques (GUO-RONG; XI-ZHENG, 2006; THUNG; LO; LAWALL, 2013; TEWARI; KUMAR; BARMAN, 2014).

FIS efficiently finds frequent itemsets in a dataset and defines the support as the number of occurrences of a subset of items (sub-itemset). A sub-itemset is considered frequent if its support is greater than a specified threshold called minimum support. Thus, support is the number of times a sub-itemset happens in the itemsets database (MAFFORT et al., 2013). Equation (4) defines support of a sub-itemset, where *I* is the number of itemsets in the database.

$$Support\ (sub\text{-}Item) = \frac{freq\ (sub\text{-}Item)}{I} \qquad \text{(Eq. 4)}$$

3.3 Evaluation of recommendation systems

Recommendation Systems (RSs) have recently become popular and have attracted a wide variety of application scenarios from business process modeling to source code manipulation. Due to this wide variety of application domains, different approaches and metrics have been adopted for their evaluation (ROBILLARD et al., 2014). RSs' quality can be defined either in terms of system-centric evaluation, which are evaluated algorithmically (e.g., precision, recall, and recall rate), or with user-centric evaluation. In user-centric evaluation, users interact with a running RS to receive recommendations. Measures are collected by asking the users (e.g., interviews or surveys), observing their behavior during use, or automatically recording interactions and subjecting system logs to various analyses (e.g., click through, conversion rate). On the other hand, with system-centric evaluation, the RS is evaluated against a pre-built ground truth dataset of opinions. Users do not interact with the RS under test but the evaluation, in terms of accuracy; it is based on the comparison between the opinion of users on items as estimated by the recommendation system and the judgments previously

collected from real users on the same items (CREMONESI; GARZOTTO; TURRIN, 2013).

Although user-centric evaluation is the only one able to truly measure the user's satisfaction on recommendations and the quality of the decision making process, conducting empirical tests involving real users (user-centric evaluation) is difficult, expensive, and resource demanding. On the other hand, system-centric evaluation has the advantage to be immediate, economical, and easier to perform on several domains and with multiple algorithms (CREMONESI; GARZOTTO; TURRIN, 2013).

System-centric quality can be measured by using (CREMONESI; GARZOTTO; TURRIN, 2013): i) Error metrics like RMSE (Root Mean Square Error) and MAE (Mean Absolute Error); or ii) Classification accuracy metrics like precision, recall, fallout, and others. Nevertheless, as real users are not involved in system-centric evaluation, there is a necessity of splitting the dataset into a training set and a test set. The dataset is usually split according to one of three main methods (CREMONESI et al., 2008): i) holdout; ii) leave-one-out; or iii) *k*-fold cross validation.

3.3.1 Dataset partitioning methods

The training data are used by one or more learning methods to come up with the model and the test data are used to evaluate the quality of the model. The test set must be different and independent from the training set in order to obtain a reliable estimate of the true error. Dataset is usually split according to one among the following methods (CREMONESI et al., 2008):

a) **Holdout**. It splits a dataset into two parts: i) a training set; and ii) a test set. These sets could have different proportions. In the setting of recommendation systems, the partitioning is performed by randomly selecting some items from all (or some of) the users. The selected items

constitute the test set, while the remaining ones are the training set. This method is also called leave-*k*-out. Many studies split the dataset into 80% training and 20% test data repeating 10 times each experiment with different training and test sets and results are averaged;

b) **Leave-one-out.** Obtained by setting $k = 1$ in the leave-k-out method. Given an active user, it is withhold in turn one rated item and the algorithm is trained on the remaining data. The withheld element is used to evaluate the correctness of the prediction and the results of the evaluations are averaged in order to compute the final quality estimate. This method has some disadvantages, such as, the overfitting and the high computational complexity. This method is suitable to evaluate the recommending quality of the model for users who are already registered as members of the system;

c) ***k*-fold cross validation.** It is a variant of the holdout method and it consists in dividing the dataset into *k* independent and equal size folds (so that folds do not overlap). In turn, each fold is used exactly once as test set and remaining folds are used for training the model. By choosing a reasonable number of folds, thus, mean, variance, and confidence interval can be computed. In evaluations with this method, researchers set *k* to 5 or 10 (JANNACH; ZANKER, 2012). This technique is suitable to evaluate the recommending capability of the model when new users (i.e., users do not belong to the model yet) join the system.

3.3.2 Error metrics

These metrics measure how a recommendation system can predict the ratings of users. Thus, if the recommendations produced are intended to predict how users rate items of interest, then root-mean-squared-error (RMSE) or mean absolute error (MAE) metrics are often used (ROBILLARD et al., 2014).

MAE is the most popular metric when evaluating the ability of a system to correctly predict a user's preference for a specific item. MAE computes the average deviation between computed recommendation scores and actual rating values for all evaluated users and all items in their testing sets. RMSE is used to put more emphasis on larger deviations (JANNACH et al., 2010).

However, MAE, RMSE, and related metrics may be less meaningful for tasks such as Finding Good Items and top-*N* recommendation, where a ranked result list of *N* items is returned to the user. Thus, the accuracy for the other items, which user will have no interest in, is unimportant (CREMONESI et al., 2008). Equations 5 and 6 correspond to MAE and RMSE respectively, where $P_{u,i}$ is the predicted ratings for *u* on item *I*, and $R_{u,i}$ is the actual rating, and *N* is the number of ratings in the test set (ZUVA et al., 2012).

$$MAE = \frac{\sum |P_{u,i} - R_{u,i}|}{N} \quad \text{(Eq. 5)}$$

$$RME = \sqrt{\frac{(P_{u,i} - R_{u,i})^2}{N}} \quad \text{(Eq. 6)}$$

Unlike MAE, large errors are not tolerated in RMSE. MAE would prefer a system that makes few errors even if they are big errors (e.g., to predict a rating as 1 when in fact it is 5) whereas RMSE would prefer a system that makes many small errors rather than few big errors (ALHARTHI, 2015).

3.3.3 Classification accuracy metrics

Classification accuracy metrics measure to what extent a recommendation system is able to correctly classify items as interesting or not (ROBILLARD et al., 2014). With classification metrics, each recommendation can be classified into four categories (CREMONESI et al., 2008): i) true positive (TP, an interesting item is recommended); ii) true negative (TN, an uninteresting item is not recommended); iii) false negative (FN, an interesting item is not recommended);

and iv) false positive (FP, an uninteresting item is recommended). That categorization is known as confusion matrix (Table 2).

Table 2 Categorization of all possible recommendations - confusion matrix
Source: Robillard et al. (2014)

	Recommended	Not Recommended	Total
Used	true positives (TP)	false negatives (FN)	Total Used
Not Used	false positives (FP)	true negatives (TN)	Total Not Used
Total	Total Recommended	Total Not Recommended	Total

The purpose of a classification task in the context of item recommendation is to identify the *N* most relevant items for a given user. Precision and recall are the two best-known classification metrics (JANNACH et al., 2010). Precision is the probability that a recommended item corresponds to the user's preferences (Eq. 7). Recall is the probability that a relevant item is recommended (Eq. 8). Thus, Precision and recall, are inversely related and in most cases, increasing the size of the recommendation set will increase recall but decrease precision (SCHRÖDER; THIELE; LEHNER, 2011). Then, we can always improve one of the metrics (recall or precision) by declining the other. For instance, the larger the number of items in recommendation lists, the recall tends to be maximal (almost all the useful items would be shown), though the precision would be as bad as the proportion of useful items (HERNÁNDEZ; GAUDIOSO, 2008). Because of mutual dependence between recall and precision, it makes sense to consider them in conjunction with two other metrics (SCHRÖDER; THIELE; LEHNER, 2011): i) miss rate (Eq. 9), the probability that a relevant item is not recommended; and ii) fallout (Eq. 10), the probability that an irrelevant item is recommended.

$$precision = \frac{TP}{TP + FP} \quad \text{(Eq. 7)}$$

$$recall = \frac{TP}{TP + FN} \quad \text{(Eq. 8)}$$

$$miss\ rate = \frac{FN}{TP + FN} \quad \text{(Eq. 9)}$$

$$fallout = \frac{FP}{FP + TN} \quad \text{(Eq. 10)}$$

These metrics are suitable for evaluating top-*N* lists of items recommended. When a recommendation algorithm predicts the top-*N* items that a user expects to find interesting, by using the recall can be computed the percentage of known relevant items from the test set that appear in the *N* predicted items (CREMONESI et al., 2008). Hence, a good algorithm should have large recall (i.e., it should be able to recommend items of interest to the user) and low fallout (i.e., it should avoid to recommend items of no interest to the user) (CREMONESI; GARZOTTO; TURRIN, 2013).

On the other hand, recall rate@*k* is another useful metrics for evaluating effectiveness of recommendation systems. Recall rate@*k* is the proportion of top-*k* recommended lists, in the set of all recommendations (for all projects) that includes at least one relevant item recommended (THUNG; LO; LAWALL, 2013). This metric has a different nature, it responds with what percentage of cases the answer was positive, i.e., measure the efficacy (MALHEIROS, 2011). Then, For computing recall rate@*k*, where *k* is the number of the recommended items, we must assign the value 1 or 0, where value 1 is assign if at least one of the *k* recommended items (e.g., API) is a member of relevant items and value 0 otherwise (THUNG et al., 2013).

4 API RECOMMENDATION METHODOLOGY

There are three major processes in recommendation systems (BIGDELI; BAHMANI, 2008): i) object data collections and representations, which consists in getting the items to be analyzed; ii) similarity decisions, which involves the calculation of distance/similarity among data; and iii) recommendation computations, which introduces and filters recommendation of relevant items.

The remainder of this chapter is organized as follows. Section 5.1 presents context and criteria data. Section 5.2 introduces the collection of data that was analyzed. Section 5.3 presents similarity decisions and the API recommendation engine. Section 5.4 shows the evaluation criteria and the evaluation strategy.

4.1 Baseline software data

Collection of baseline software data is one of the major processes of RSs. These baseline software data were the Java software obtained from Sourcerer[1] dataset, which is part of the source code repository UCI (University of California, Irvine) (BAJRACHARYA; OSSHER; LOPES, 2009; LOPES et al., 2010). The repository contains about 18,826 (~13,241 non-empty) software from Apache[2], Java.net[3], Google Code[4], and SourceForge[5]. That repository takes up 400 GB of HD (Hard Disk) and is periodically updated and the previous content archived (BAJRACHARYA; OSSHER; LOPES, 2014).

Relationship between software and software categories was part of the requirements of this work. Thus, we considered the categories established by the repository itself in order to avoid subjective classification criteria. In Sourcerer,

[1] http://sourcerer.ics.uci.edu/
[2] http://apache.org/
[3] http://java.net/
[4] https://code.google.com/
[5] http://sourceforge.net

there were software from five repositories; we established some criteria for the appropriate choice. Consequently, we took into account the level of granularity of provided categories and the corresponding number of available software on Sourcerer. Besides, we used SourceForge because it had 9,969 (~6,632 non-empty) in Sourcerer dataset and provided 10 main categories. Thus, we decided to get relevant software. In this context, was determined as relevant, software that:

a) Were developed in Java and Eclipse IDE-platform;
b) Used at least 2 APIs;
c) Were independent (not embedded software);
d) Belonged exclusively to one SourceForge category;
e) Had been registered for at least 4 years;
f) Last date of maintenance must be between 2010 and 2015.

Briefly, after filtering the data by the established criteria, we obtained 830 relevant software and we put them in a main software repository called *M*. After this, we manually classified these into SourceForge categories and saved in spreadsheets the information of:

a) *Batch*: Number for the directory structure defined by Sourcerer dataset where batches 40 to 54 contents SourceForge software;
b) *Id*: Number for the subdirectory structure defined by Sourcerer. Every batch contains many id subdirectories with individual software inside;
c) *Name_batch*: Software name located in the file *project.properties* in every software directory;
d) *Name_SourceForge:* Software name in the SourceForge repository;
e) *Category:* Category defined by SourceForge;
f) *ReleaseDate:* Release date provided by SourceForge;
g) *Intended audience:* Specific target audience assigned by Sourceforge for each software.

Thereafter, we used a plug-in for the Eclipse IDE known as AnyEdit tools[6] for setting and saving the information and the relation of every software with the corresponding category. We used this plug-in mainly because we can store definitions of working sets to `.wst` files and import them in any other workspace. Thus, we made the manual assignment of each SourceForge category just once. We set working sets as categories containing corresponding software (Figure 3).

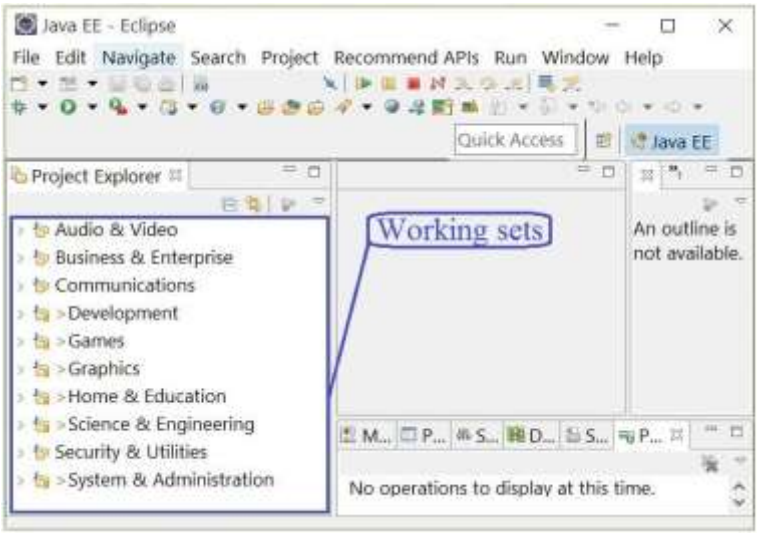

Figure 3 SourceForge categories view as working sets in the Eclipse IDE

In addition, for automating the correct selection of software related to the target software of recommendation, we developed the corresponding modules in a plug-in (Chapter 6).

4.2 Data collection

The data collection process is one of the three major processes in recommendation systems (BIGDELI; BAHMANI, 2008). In this context, it consists in getting the items to be analyzed; therefore, items are APIs.

[6] http://andrei.gmxhome.de/anyedit/index.html

Within Java software, there are different ways to capture information about APIs, some of them are: i) the corresponding binary files (`.jar` files); ii) the `import` statement in `.java` file where the corresponding class term should be disregarded; and iii) the declared tags of `<dependency>` via `pom.xml` files in Maven software.

In this work, we decided to capture APIs through `import` statement in the `.java` files. That statement allows reusing existing features and there are two ways to reference them: i) By the entire package using the `*` character referencing all of the members (class or interface) contained in the package (e.g `.import graphics.*`); or ii) By package members (class or interface) using their simplified name (e.g `.import graphics.Circle`). In those two examples, the same API was referenced: Graphics.

At first, packages appear to be hierarchical, but they are not. For example, the Java API includes a `java.awt` package, a `java.awt.color` package, a `java.awt.font` package, and many others that begin with `java.awt`. However, the `java.awt.color` package, the `java.awt.font` package, and other `java.awt.xxxx` packages are not included in the `java.awt` package. The prefix `java.awt` (Java Abstract Window Toolkit) is used for a number of related packages to make the relationship evident, but not to show inclusion. For instance, `java.awt.*` imports all types in the `java.awt` package, but it does not import `java.awt.color, java.awt.font`, or any other `java.awt.xxxx` packages. Thus, to use the classes and other types in `java.awt.color` as well as those in `java.awt`, we must import both packages with all their files: `import java.awt.*` and `import java.awt.color.*` (THE JAVA TUTORIALS: ORACLE, 2014).

Therefore, we considered all import statements by removing their package members (classes or interfaces) statements because a Java API is a collection of packages. For example, in the import statement `org.apache.log4j.Logger`,

the API package is `org.apache.log4j` and `Logger` is a class from that package (Figure 4). Other important decision was to differentiate between own software packages and external APIs packages in the import statements.

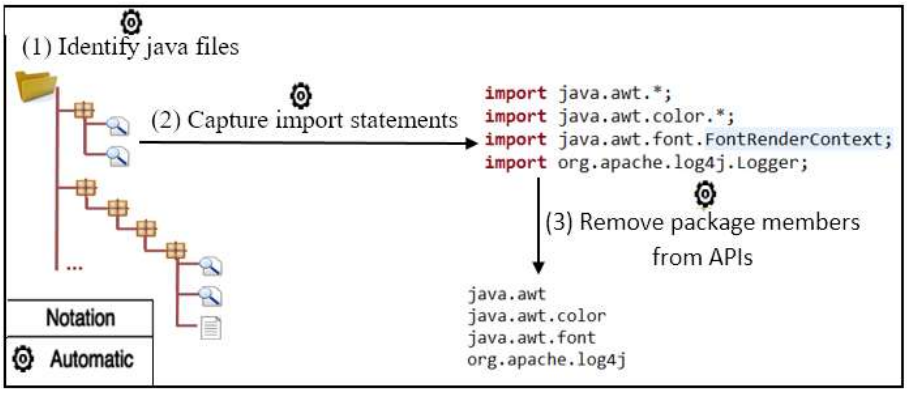

Figure 4 Example of APIs extraction process

In order to automate the data collection methodology, we developed a corresponding module in a plug-in (Chapter 6).

4.3 Recommendation engine

The recommendation engine considers one software category and consists on strategies and techniques used to obtain the relevant APIs considering two stages of software development:

Stage A: API Recommendation for Software Engineers in initial stage of software development (not using APIs);

Stage B: API Recommendation for Software Engineers in advanced stage of software development (using some APIs).

Regarding the context of Stage A, recommendation techniques were not a good choice because there were not APIs to link between the target software provided by the Software Engineer and the model software (*MS*), i.e., software dataset of recommendation system; in our evaluation, it is software in training sets. In recommendation systems, it is known as the cold-start problem (Section

3.2). However, in order to recommend APIs for Software Engineers in initial stage of software development (Stage A), we decided to consider the popularity of APIs. Then, we decided to use the Frequent Itemset mining technique (FIS) (Section 3.4). In FIS, we must define the threshold called minimum support. After that, most common APIs in the model software for each category were found. Next, the results are related to categories of the target software for recommending APIs.

Regarding the Stage B context, we determined to use Collaborative Filtering technique (CF) in conjunction with the Frequent Itemset mining (FIS) technique used in Stage A. That decision implied in using the Nearest Neighbors approach (Section 3.3) for computing similarity between the target software for API recommendation and model software found with same categories of target software based on their set of commonly used APIs. For the similarity computation, we decided to use the well-known metric Jaccard (Section 3.3.3). Hence, we must provide the number of NN to be considered. Thus, we decided to vary NN in a top-10 of recommendation and we chose the one with best values of evaluation metrics. Following, in order to find the relevant APIs from similar software systems, we computed the popularity for APIs used by those software and the APIs did not used in target software. Consequently, we sorted them in decreasing order, generating a possible list of APIs to recommend. Afterwards, we used FIS mining technique to generate a list of possible APIs to recommend. Then, we made the union of possible lists of recommendations generated by CF and FIS according to the value of popularity. Finally, we computed the top-*N* of recommendation because this recommendation will be more useful for Software Engineers to have some APIs to consider into their development than having a large list with many APIs. Thus, in this stage, we expected to have more "accuracy" on final lists of APIs recommended due to the use of two techniques. Finally, for automating the recommendation engine, we developed a corresponding module for each stage in a plug-in (Section 6.3).

4.4 Evaluation of the recommendation system

We decided to evaluate the recommendation methodology and the plug-in using a system-centric evaluation and without real users' interaction because conducting empirical tests involving real users is difficult, expensive, and resource demanding. On the other hand, system-centric evaluation has the advantage to be immediate, economical, and easy to perform on several domains and with multiple algorithms (CREMONESI; GARZOTTO; TURRIN, 2013). For that, we used precision, recall, and recall rate (Section 4.3).

To perform evaluation without real users' interaction, we decided to use *k*-fold cross validation where data are divided in *k* subsets. We used $k = 5$ then, one of the subsets (20% of data) is used as test data and the remaining (*k*-1, i.e., 80% of data) as training data (Figure 5). We repeated this process *k* times, i.e., five iterations with each of possible subset of test data. In this process, we saved working sets of data set partition for every category in `.wst` files.

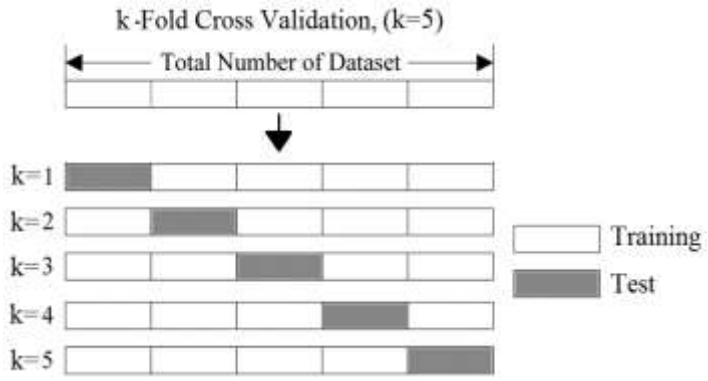

Figure 5 Example of data partition in *k*-Fold Cross validation

We evaluated Stage A and Stage B through *k*-fold cross validation for every category dataset with a different simulation strategy. In Stage A, we must consider that target software do not use any APIs at all. In order to simulate the target software behavior of Stage A, all APIs (100%) were removed and saved in

an `.xml` file. Those APIs are the relevant, thus, the ones expected to be included in the recommendation list. In Stage B, target software used some APIs already. Thus, a percentage of APIs from target software must be removed and saved in a `.xml` file. Those APIs are the relevant and the ones expected to be included in recommendation list. In this context, we decided to randomly remove 50% of APIs from every target software of recommendation. As this is a randomized process, we did five replicates of recommendation for every target software, i.e., for each software from test set in every iteration.

In the recommendation tests, evaluation was performed by applying the precision, recall, and recall rate metrics. Independently of the stage, the expectation was our recommendation methodology would be able to make useful recommendations for Software Engineers. Thus, we expected that APIs removed, i.e., relevant APIs, should appear in the recommendation lists causing higher precision, recall, and recall rate. Finally, for automating evaluation through users' simulation, we developed a corresponding module in a plug-in (Section 6.4).

5 PLUG-IN DEVELOPMENT

In order to automate the API recommendation methodology proposed in Chapter 5, we developed a plug-in for the Eclipse IDE. We used a DELL[7] computer with Windows 10 system, Intel® Core ™ i7 2.4GHz, and 16GB RAM. Moreover, we used the Eclipse Java EE IDE for Web Developers release 1a (4.4.1) of Luna version[8] and the Java infrastructure known as JDT Core[9].

In Figure 6, we displayed the automation strategy of the recommendation methodology, using notation from SPEM (Software & Systems Process Engineering Metamodel) and BPM (Business Process Management). We achieved that automation through the development of three main modules: i) software data module, that consists in the selection of the software related to the categories of the target software; ii) data collection module, that involves extraction and manipulation of APIs; and iii) recommendation module, that comprises the development of the recommendation techniques and the obtainment of recommendation lists.

In addition, we developed an extra module of simulation for evaluating the methodology in an automatic way and without real users' interactions because of the advantages to be immediate, economical, and easier to perform on several domains and with multiple algorithms (CREMONESI; GARZOTTO; TURRIN, 2013).

For explaining the development of every module, this chapter is organized as follows. Section 6.1 shows the module for selecting software. Section 6.2 presents the module for data. Section 6.3 exposes the recommendation module. Section 6.4 explains the simulation module.

---

[7] http://www.dell.com/
[8] http://www.eclipse.org/downloads/packages/release/Luna/
[9] https://eclipse.org/jdt/core/

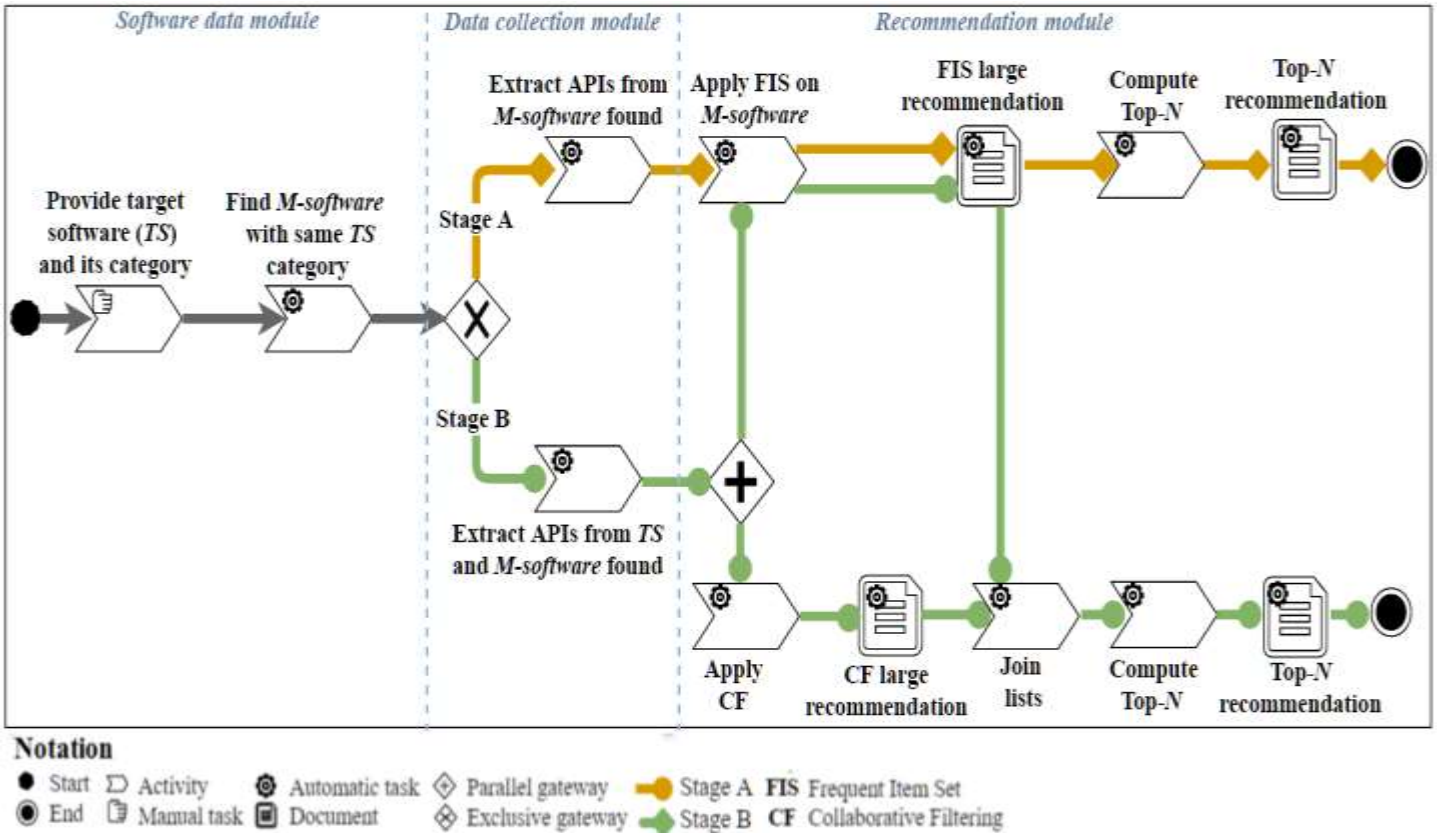

Figure 6 API recommendation strategy for Software Engineers in initial and advanced stage of software development (Stage A and Stage B, respectively)

5.1 Software data module

The software data module depends on two main activities (Figure 6), the first one is manual and the second one is automatic. The manual activity consists in the Software Engineer to select the Target Software (*TS*) and its categories (*TScategories*). Thereafter, the second activity is automatically triggered. That activity consists on automatically finding the software from the model software (*MS*), i.e., the software dataset of recommendation system, in our evaluation; it is software in training sets but with same *TScategories*.

We displayed the interface to the manual activity where the Software Engineer can select the *TS* and its categories (Figure 7). That interface is the same for Stage A and for Stage B of recommendation.

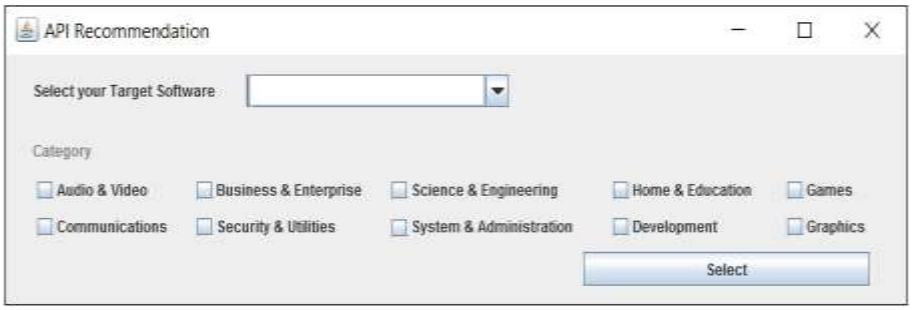

Figure 7 Interface for selection of Target Software and its categories

After selecting the target software and its category, the automatic task is triggered for finding model software with same categories. Algorithm 1 displays a simplified version of that automation. Besides, in this module, we used an API known as `org.eclipse.ui` for capturing and manipulating working sets' information.

5.2 Data collection module

In the proposed methodology, we decided to capture APIs through `import` statements (Section 5.2). This module corresponds to automation of API extraction of software. As displayed in Algorithm 2, the core of this module consist in API extraction for every software (*SW*), either target software (*TS*) or model software found (*foundSoftware*) from model software training set (*MS)*. Besides, we added some features, for example: i) removal of statements corresponding to the package members (i.e., classes or interfaces) as showed before in Figure 4; and ii) the distinction between the own software packages and the external APIs packages in the import statements.

Algorithm 1 Simplified version of software data selection algorithm

```
1: Input:
2: TS: set of target software of API recommendation
3: TScategories: set of categories for every software of TS
4: MS: training set of Model Software
5: Output:
6: foundSoftware: Set of M-found software with same categories of TS
7: Method:
8: Let foundSoftware = ∅
9: for each T-software ∈ TS do
10:   for each T-category ∈ TScategories do
11:     for each M-software ∈ MS do
12:        if T-category = M-category then
13:           M-found = M-software
14:           add M-software and T-category to foundSoftware
15:        end if
16:     end for
17:    end for
18: end for
19: return foundSoftware
20:
```

5.3 Recommendation module

This is the main module of the plug-in. It consists in the automation of the API recommendation techniques, regarding that we used them according to the criteria of every stage of recommendation (Section 5.3). We showed recommendation module for Stage A and Stage B in following sections.

5.3.1 Recommendation based on Stage A

This module corresponds to the strategy for recommending APIs attending Stage A criteria (Figure 6), i.e., where API Recommendation is made for Software Engineers in initial stage of software development (do not use APIs) that considered one software category.

Algorithm 2 Simplified version of API collection algorithm

```
Input:
SW: software set, either TS, MS, foundSoftware, or any other set
of sotware.
Output:
sw-APIs:set of APIs used by every sw ∈ SW where sw can be T-
software or M-software depending on SW.
Method:
Let sw-APIs = ∅
for each sw ∈ SW do
   for each package ∈ sw-Packages do
      add package to ownPackages
   end for
   for each package ∈ ownPackages do
      for each javaFile ∈ package do
         for each API ∈ javaFile
            if API ∉ ownPackages then
              remove Class or Interface statements from API
              add API to sw-APIs
            end if
         end for
      end for
   end for
end for
return sw-APIs
```

In this module, the core is the Frequent Itemset mining (FIS) technique in order compute the frequency of APIs. As exposed in Section 3.4, a technique to determine that frequency was by finding the most common APIs into the training dataset of software from selected categories of the target software of recommendation (*TS*), i.e., applying Equation 4 for computing the support value of each API used in *foundSoftware* from *TScategories*. Consequently, an API is considered frequent in a category if its *support* was greater than or equal to the specified threshold called *minSupport*. Algorithm 3 shows a simplified version of

the implemented process from FIS technique in Stage A, where the input is the threshold value and the outputs are the frequent APIs for selected *TScategories*.

Algorithm 3 Frequent APIs mining for Stage A

```
1: Input:
2: MinSupport: treshold for API occurrence in a subset
3: Output:
4: APIsFreq: Set of APIs with Support and TScategories
5: Method:
6: Let APIsFreq = ∅
7: for each T-category ∈ TScategories do
8:    for each M-found ∈ foundSoftware do
9:       for each API ∈ sw-APIs from M-found do
10:         frequency[API] = frequency[API]+1
11:      end for
12:   end for
13:   support[API] = frequency[API] ÷ number of foundSoftware
14:   if support[API] ≥ MinSupport then
15:      add API and support[API] and T-category to APIsFreq
16:   end if
17: end for
18: return APIsFreq
```

Other main attribute for this module is the size *N* of the top-*N* recommendation list since after finding all the frequent APIs for each *T-category* ∈ *TScategories* those APIs are sorted in decreasing order. Later, two main recommendation lists are generated, one with all of the API recommendations and other with the most *N* frequent APIs in a top-*N* list. In Figure 8, we showed the interface of the plug-in for setting the input data where the tag *Min. Threshold value* is the *minSupport* and the tag *Quantity of APIs for* top-*N* is *N*.

Figure 8 Interface for setting API recommendation based on Stage A

5.3.2 Recommendation based on Stage B

This module corresponds to the strategy for recommending APIs meeting Stage B criteria (Section 5.3), i.e., where an API Recommendation was made for Software Engineers in advanced stage of software development (which use some APIs) that considered one software category. Consequently, recommendation module is composed by two techniques: i) Frequent Itemset (FIS); and ii) Collaborative Filtering (CF).

5.3.2.1 Frequent itemset mining technique - (FIS)

The FIS technique is the same technique used in Stage A, although we applied some modifications to the algorithm because of the context of Stage B, where Target Software (*TS*) of recommendation already use some APIs. Thus, those APIs are not considered in the recommendation lists. Algorithm 4 displays a simplified version of the FIS technique used as part of the recommendation module regarding Stage B where the input is the threshold value and the outputs are the frequent APIs for selected *TScategories*. In Algorithm 4, lines 14 to 17 present the main differences between both FIS algorithms (3 and 4), where APIs used in *TS* are not considered for recommending; thus, they can be removed from possible frequent APIs to be recommended.

After finding all the frequent APIs for each *T-category* $\in$ *TScategories*, they are sorted in decreasing order. Thereafter, a list with all of API recommendations is generated to subsequently make the corresponding union with the recommendation list generated by the Collaborative Filtering technique.

5.3.2.2 Collaborative filtering recommendation technique - (CF)

The aim in this technique is to recommend APIs based on those APIs used in similar software found by the nearest neighbor collaborative filtering algorithm.

Algorithm 5 displays a simplified version of the CF module developed as part of the recommendation engine regarding Stage B where the input attribute is *k*, i.e., the number of nearest neighbors to consider.

Algorithm 4 Frequent APIs mining for Stage B

```
 1: Input:
 2: MinSupport: treshold for API occurrence in a subset
 3: Output:
 4: APIsFreq: Set of APIs with Support and TScategories
 5: Method:
 6: Let APIsFreq = ∅
 7: for each T-category ∈ TScategories do
 8:    for each M-found ∈ foundSoftware do
 9:       for each API ∈ sw-APIs from M-found do
10:          frequency[API] = frequency[API]+1
11:       end for
12:    end for
13:    support[API] = frequency[API] ÷ number of foundSoftware
14:    for each T-software ∈ TS do
15:       if API ∈ sw-APIs from T-software then
16:          remove API from support[API]
17:       end if
18:    end for
19:    if support[API] ≥ MinSupport then
20:       add API and support[API] and T-category to APIsFreq
21:    end if
22: end for
23: return APIsFreq
```

In this algorithm, the similarity of every pair of software is based on their set of commonly used APIs, e.g., the similarity value computed in lines 14 and 15. Thereafter, the *k* nearest software (*kNN*) are selected and their API frequency is computed in order to find the most popular APIs in the most similar software. Afterwards, APIs used in *TS* are not considered for recommending; thus, they are removed from possible frequent APIs to be recommended found. Finally, *kNN* for every *T-software* ∈ *TS* are returned along with a corresponding set of possible frequent APIs to recommend. Regarding that, those possible APIs are sorted by frequency value, i.e., by *support*.

After computing all the frequent APIs from similar software (*kNN*) from each *T-category*, those APIs are sorted in decreasing order. Thereafter, a list of

API recommendations is generated to subsequently make the union with the list generated by the FIS technique.

Algorithm 5 Collaborative technique for Stage B

```
Input:
k: number of NN (Nearest Neighbors) to consider
Output:
kNN: set of KNN-software to TS with corresponding sim value and
SwAPI with support value sorted in decreased order. It considers
just APIs available to recommend.
Method:
Let kNN = ∅
for each T-software ∈ TS do
   for each T-category ∈ TScategories do
      for each M-found ∈ foundSoftware do
         sim = similarity(sw-APIs from T-software and sw-APIs
         from M-found)
         add M-found and sim and T-category to allNN
      end for
      Sort allNN by decreasing order of sim value
      kNNsoftware = Select K nearest software from allNN
      for each API ∈ sw-APIs from kNNsoftwaredo
         frequency[API] = frequency[API]+1
      end for
      support[API] = frequency[API] ÷ number of kNNsoftware
      if API from kNNsoftware ∉ sw-APIs from T-software then
         add kNNsoftware and sim and support[API] to kNN
      end if
   end for
end for
return kNN
```

5.3.2.3 Junction of FIS and CF lists

After applying both techniques used for recommendation as exposed, two possible list of recommendation are generated. Both lists are already sorted by *support* value and there is made the union between the lists. For that union, we assigned an equitable weight of 50% of relevance for every strategy (Eq. 11).

$$support[API](0.5 \times FISsupport[API]) + (\,0.5 \times CFsupport[API]) \quad \text{(Eq. 11)}$$

In addition, other attribute must be considered, that is the size *N* of the top-*N* lists since two final recommendation lists are generated to the Software

Engineers with *TS* in advanced stage of development, one with the union of all API recommendations and the other with the top-*N* list of recommendation.

In Figure 9, we showed the interface of the plug-in for setting input data where the tag *Number of Nearest Neighbors* is *k*, the tag *Min. Popularity Threshold Value* is *minSupport*, and tag *Quantity of APIs for* top-*N* is *N*.

Figure 9 Interface for setting API recommendation based on Stage B criteria

On the other hand, either for Stage A or Stage B of recommendation, the plug-in exposes the top-*N* of recommended APIs to the Software Engineers through the interface and in `.xml` files. The `.xml` file contents the APIs with the corresponding support value. In Figure 10, we presented a toy example of a top-10 recommendation through the interface where APIs are sorted in decreasing order, i.e., the first recommended API is the one with major *support* value.

Figure 10 Interface for showing top-*N* API recommendations

5.4 Simulation module

We developed a feature for simulating the user's interaction based on the strategy presented in Figure 11, where we used notation from SPEM (Software &

Systems Process Engineering Metamodel) and BPM (Business Process Management). In that evaluation strategy, we considered both stages (A and B) and simulated the users' interaction by removing APIs (defining them as relevant) to subsequently compare them with the APIs recommended by the plug-in. We made that simulation based on each stage and considered the dataset partition of 5-fold cross validation for every category.

For Stage A, all APIs are removed to simulate Software Engineers in initial stage of software development. On the other hand, for Stage B, percentages (from 0.01 to 0.9) of APIs to remove can be provided to simulate Software Engineers in advanced state of software development. In this project, we removed half of APIs, i.e., 0.5 of APIs are removed in every simulation. In addition, we varied attributes like threshold (*minSupport*) value for FIS technique and number of nearest neighbors (*k*) in order to find the best for every category and use it to evaluate every stage of the API recommendation methodology.

Afterwards, for evaluating the quality of our API recommendation methodology, we applied recall, precision, and recall rate (evaluation metrics) based on the `.xml` files of APIs and the recommended, i.e., based on APIs removed defined as relevant and the two recommendation lists generated (one list with large recommendations and the other list with the top-*N* recommendations). Finally, this module generates those metric results in `.xls` files for further manual analysis. Nevertheless, it is important to consider that in Stage B, we did five replicates of the recommendation test for each target software since the APIs removal was randomized (Section 5.4). Thus, in those tests we averaged results in order to evaluate recommendations made for every target software in each iteration of the 5-fold cross validation for certain category.

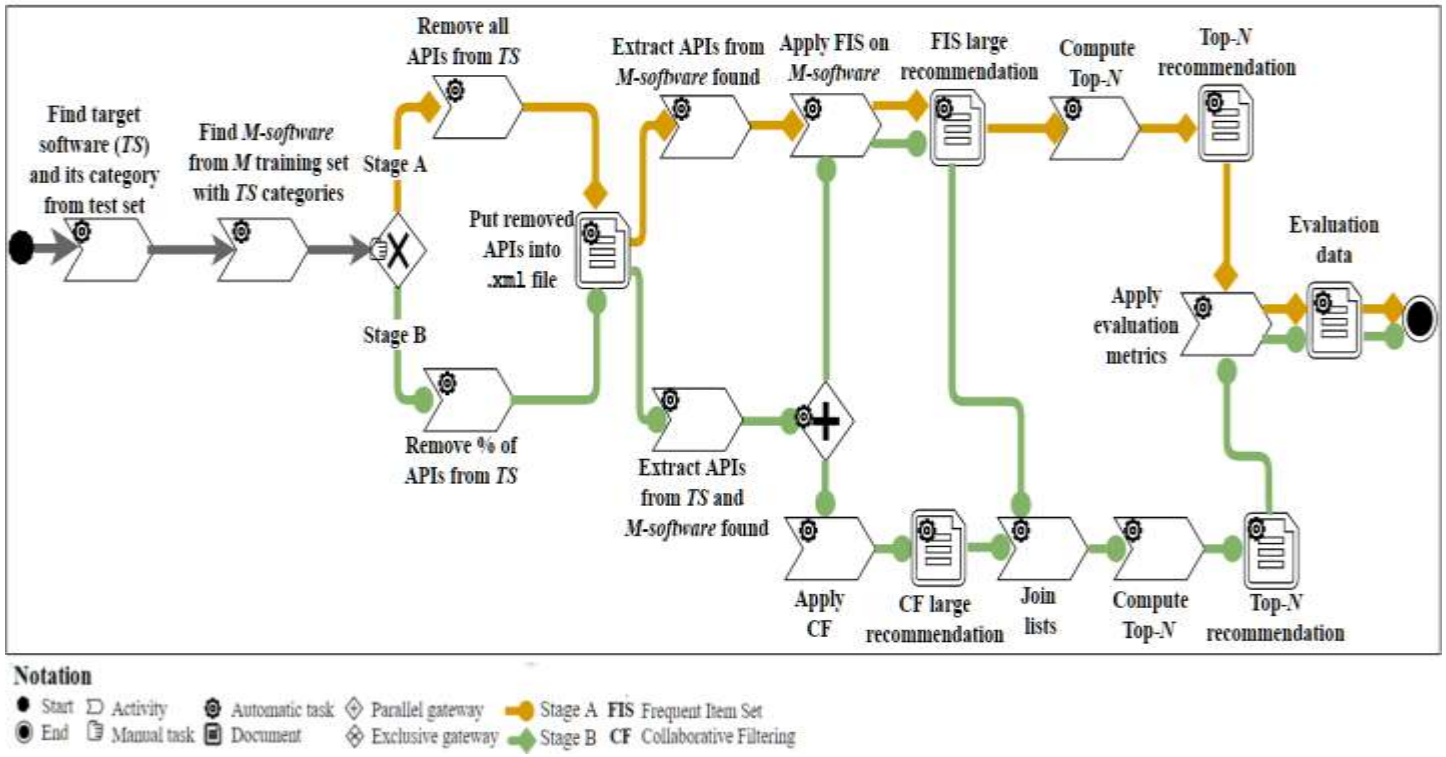

Figure 11 Evaluation strategy for the proposed API recommendation methodology

6 RESULTS

After applying the criteria over the Sourcerer dataset (Section 5.1), we obtained 830 relevant Java software that belong exclusively to one of the 10 SourceForge categories. Then, we presented the acronym and the number of software found for every category (Table 3).

Table 3 Baseline software data for every SourceForge category

CATEGORY	ACRONYM	NUMBER OF SOFTWARE
Audio & Video	AV	35
Business & Enterprise	BE	35
Communications	C	50
Development	D	510
Games	GA	70
Graphics	GR	20
Home & Education	HE	20
Science & Engineering	SE	50
Security & Utilities	SU	5
System Administration	SA	35
TOTAL		830

Although, we could not consider two categories in this work: i) Security & Utilities because of the few number of software; and ii) Development because of the large number of software, i.e., when importing software, they overflowed not just our plug-in but also the Eclipse IDE. Consequently, in this Chapter, we presented results of evaluating our recommendation methodology for the eight remaining categories, i.e., using just 315 software. Besides, we discussed those results in Chapter 8.

The remainder of this Chapter is organized as follows. Section 7.1 presents results for category Audio & Video. Section 7.2 shows results for category Business & Enterprise category. Section 7.3 displays results for Communications category. Section 7.4 exposes results for Games category.

Section 7.5 shows results for Graphics category. Section 7.6 presents results for Home & Education category. Section 7.7 exposes results for Science & Engineering category. Section 7.8 presents results for System Administration category. Finally, Section 7.9 presents results for all the eight categories.

6.1 Audio & Video category

We found 35 relevant Java software in Sourcerer dataset corresponding exclusively to Audio & Video SourceForge category. In Table 4, we showed those software and their number of APIs.

Table 4 Baseline data for Audio & Video category

#	SOFTWARE NAME	NUMBER OF APIs	#	SOFTWARE NAME	NUMBER OF APIs
1	JOPFilter	5	19	JessTab	20
2	JFreedbClient	6	20	JAud	22
3	PrismiqWeb	7	21	Amplitude	23
4	musole	8	22	ChordAssist	23
5	com.sorox.eplug	9	23	JXM	24
6	SourceForge	9	24	Sears	24
7	mp3tagmaster	13	25	protux	25
8	jlastfm	14	26	JideoGuard	28
9	jtag	14	27	Mr. Random	28
10	soundbot-current	14	28	FScape	31
11	jMp3Lib	15	29	Geeboss	33
12	CoreMP3	16	30	JTagEditor	36
13	Leipzig	16	31	plarpebu	36
14	playlister	16	32	JSynthLib-CVS	41
15	TetraHead	17	33	Meloncillo	43
16	dubman	18	34	musicminer	70
17	podcaster	18	35	jajuk	106
18	jbuzzer	20			

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Audio & Video category regardless of the stage of software development, we did the 5-fold cross validation (Figure 12). Hence, in every iteration, we used seven different software (i.e., 20% of data) as test set and 28 software (i.e., 80% of data) as training set. In that partitioning, we

avoided overlapping, i.e., every software appears just once in test set. In this partitioning process, we saved `.wst` files of every iteration.

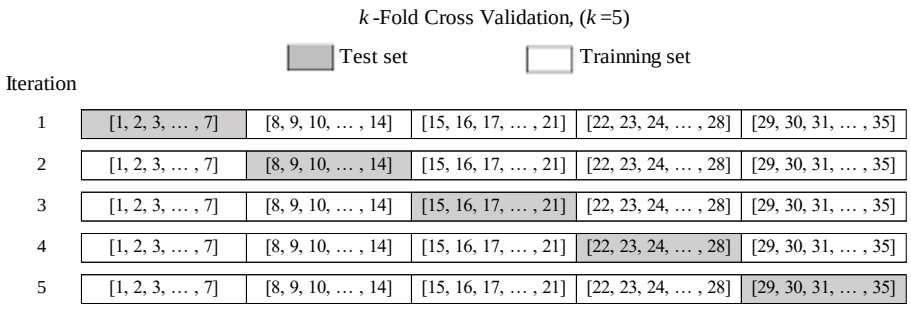

Figure 12 Audio & Video category dataset partition

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation methodology in every stage of API recommendations. We exposed these results as follows.

6.1.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 5). From 0.1 to 0.4, *minSupport* presented same results with the highest recall. Hence, we

chose *minSupport* = 0.1 as the "best" *minSupport value* for FIS technique in Audio & Video category, since choosing other *minSupport* value did not mean any significant improvement either for precision or recall rate values.

Table 5 Effect of varying *minSupport* value in top-10 lists of APIs recommended in Stage A of Audio & Video category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.375	0.723	1.000
0.2	0.375	0.723	1.000
0.3	0.375	0.723	1.000
0.4	0.375	0.723	1.000
0.5	0.372	0.723	1.000
0.6	0.303	0.792	1.000
0.7	0.270	0.837	1.000
0.8	0.179	0.871	1.000
0.9	0.119	0.906	1.000

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the Audio & Video category and in Stage A of development when receiving large lists of APIs recommended as well as when receiving top-*N* list of APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Table 6). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 13.

Table 6 Results of evaluation metrics for large lists of APIs recommended in Stage A of Audio & Video category.

recall	**precision**	**recall rate**
0.611	0.306	1.000

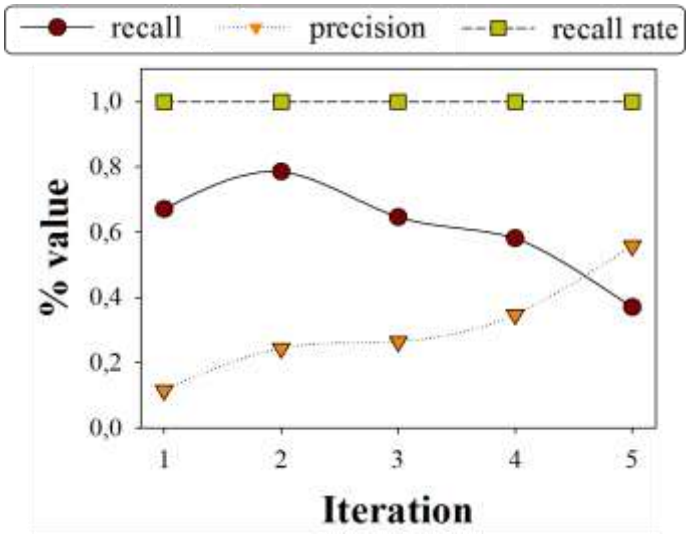

Figure 13 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of Audio & Video category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 7). In addition, for analyzing every iteration, we exposed the results of precision (Figure 14), recall (Figure 15), and recall rate (Figure 16) for each iteration.

Table 7 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of Audio & Video category.

N	**recall**	**precision**	**recall rate**
1	0.062	1.000	1.000
3	0.157	0.914	1.000
5	0.242	0.880	1.000
7	0.301	0.804	1.000
10	0.375	0.723	1.000
13	0.425	0.642	1.000
15	0.446	0.590	1.000
17	0.472	0.551	1.000
20	0.490	0.494	1.000

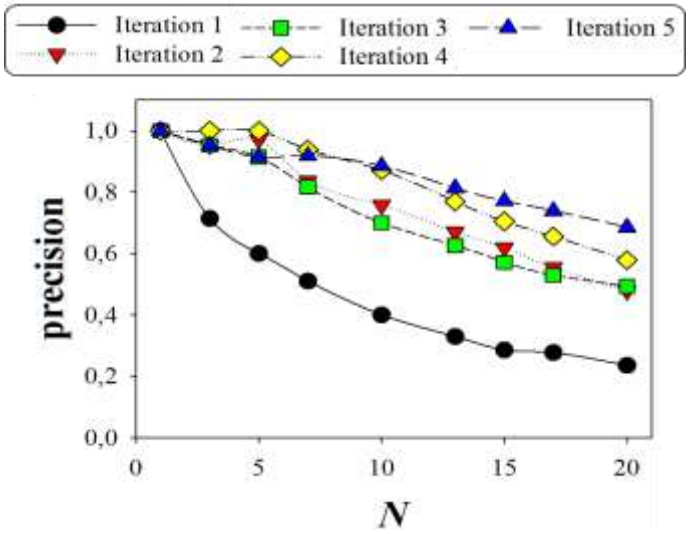

Figure 14 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Audio & Video category.

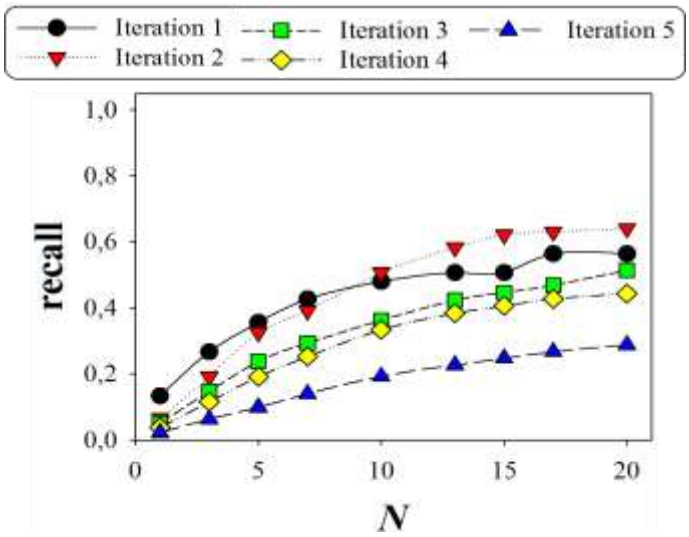

Figure 15 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Audio & Video category.

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. In Table 8, we showed number of recommended

APIs, their name and their frequency value, where frequency is the number of times the API was recommended over the 5 iterations.

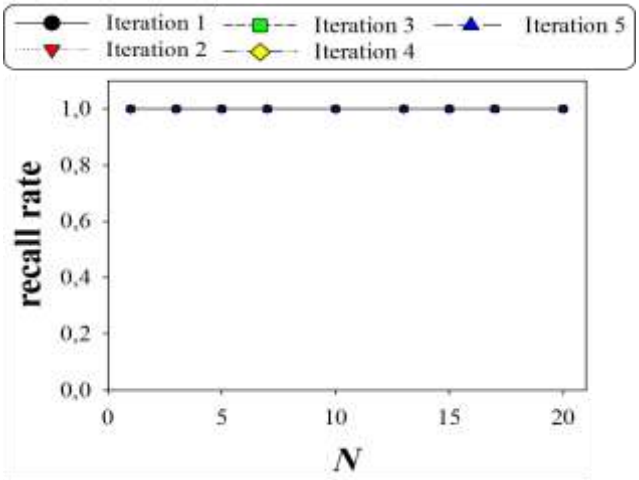

Figure 16 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Audio & Video category.

Table 8 APIs recommended in top-20 lists in Stage A of Audio & Video category.

#	API	Frequency	#	API	Frequency
1	java.awt.dnd	1	13	javax.swing.event	5
2	java.sql	2	14	java.awt	5
3	java.security	2	15	java.text	5
4	javax.swing.text	2	16	java.net	5
5	java.beans	3	17	java.util.regex	5
6	javax.sound.sampled	3	18	java.io	5
7	javax.swing.tree	3	19	java.awt.datatransfer	5
8	java.awt.image	4	20	java.awt.event	5
9	java.lang.reflect	5	21	javax.swing.border	5
10	java.util	5	22	javax.swing.filechooser	5
11	java.awt.geom	5	23	junit.framework	5
12	javax.swing.table	5	24	javax.swing	5

6.1.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-

fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found for FIS technique in Stage A, since we used the same data sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 9) and we found that varying *k* in 20 presented the best recall and precision values. Consequently, we chose *minSupport* = 0.10 for FIS technique and *k* = 20 for CF technique in Audio & Video category.

Table 9 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of Audio & Video category.

k	**recall**	**precision**	**recall rate**
5	0.431	0.447	0.971
10	0.432	0.445	0.971
15	0.432	0.446	0.971
20	0.434	0.449	0.971
25	0.429	0.442	0.971

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in Audio & Video category and Stage B of development when receiving large lists

of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In the case of top-*N*, we evaluated the effect of varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations and replicates of the 5-fold cross validation (Table 10). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 17.

Table 10 Results of evaluation metrics for large lists of APIs recommended in Stage B of Audio & Video category.

recall	precision	recall rate
0.601	0.232	0.989

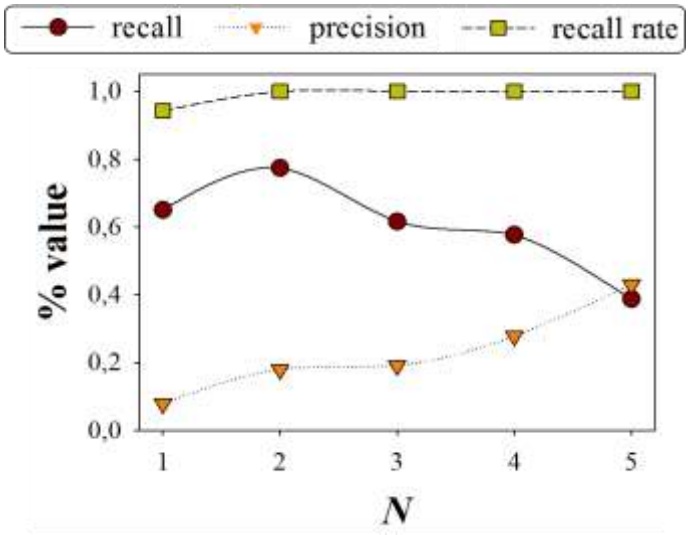

Figure 17 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of Audio & Video category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from

iterations of the 5-fold cross validation (Table 11). In addition, for analyzing every iteration, we exposed the average results of precision (Figure 18), recall (Figure 19), and recall rate (Figure 20) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates, since the removal of the 50% of APIs was randomly (Section 5.4).

Table 11 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of Audio & Video category.

N	**recall**	**precision**	**recall rate**
1	0.106	0.931	0.931
3	0.240	0.773	0.954
5	0.322	0.645	0.971
7	0.394	0.569	0.977
10	0.438	0.454	0.977
13	0.469	0.382	0.977
15	0.485	0.344	0.977
17	0.508	0.320	0.977
20	0.533	0.289	0.977

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. As in Stage B of API recommendation we made five replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value in every iteration. In this category, we found that replicates 4, 3, 5, 3, and 1 presented best recall values for iterations 1, 2, 3, 4, and 5 respectively. In Table 12, we showed the number of APIs recommended, the APIs, and their frequency value, where frequency is the number of times the API was recommended in a top-20 list over the 35 target software.

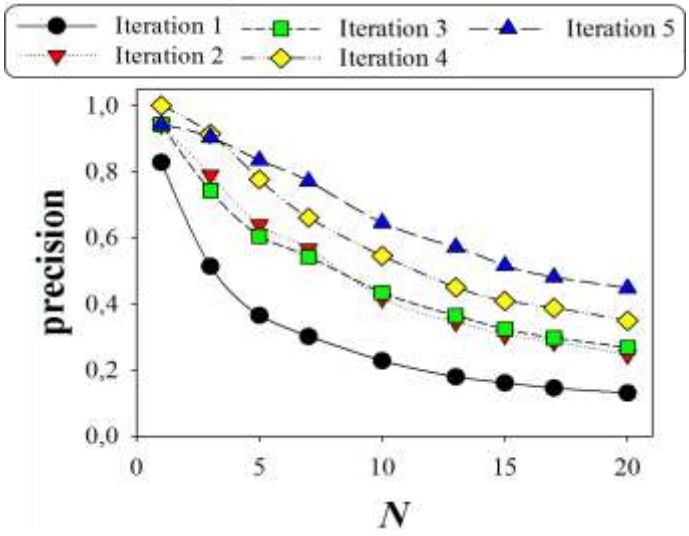

Figure 18 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Audio & Video category.

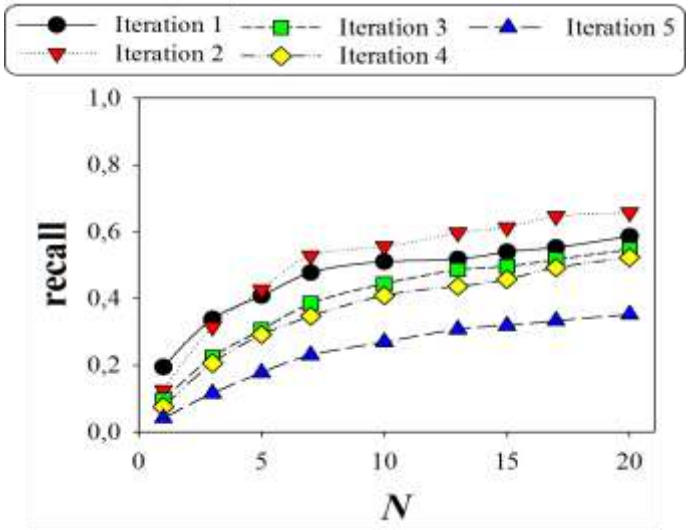

Figure 19 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Audio & Video category.

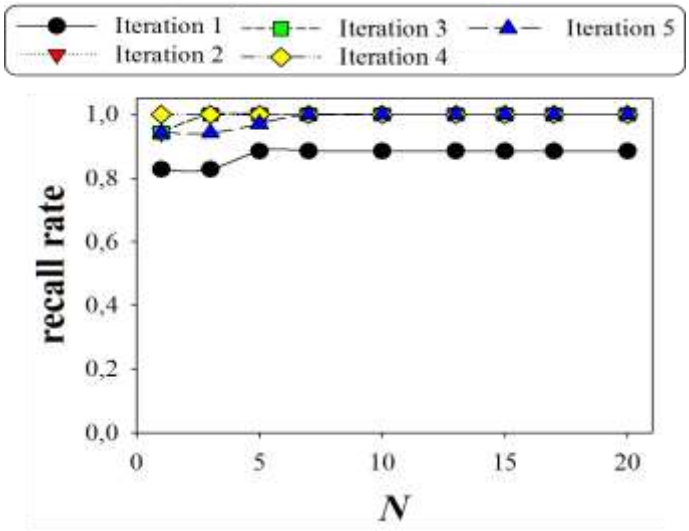

Figure 20 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Audio & Video category.

Table 12 APIs recommended in top-20 lists in Stage B of Audio & Video category

#	API	Frequency	#	API	Frequency
1	javax.xml.parsers	2	19	java.awt	23
2	javax.xml.transform	2	20	javax.swing.text	24
3	javax.xml.transform.stream	3	21	java.util	25
4	java.nio	5	22	java.security	25
5	org.xml.sax	5	23	javax.swing.filechooser	26
6	java.lang	5	24	javax.swing.event	26
7	java.math	6	25	javax.swing.tree	26
8	java.nio.channels	7	26	java.net	27
9	javax.swing	8	27	java.lang.reflect	27
10	java.awt.font	10	28	javax.sound.sampled	28
11	java.util.zip	13	29	java.beans	29
12	org.apache.log4j	14	30	javax.swing.table	29
13	java.awt.dnd	16	31	java.util.regex	30
14	java.util.prefs	18	32	java.awt.geom	30
15	java.io	20	33	java.awt.datatransfer	30
16	java.sql	21	34	java.awt.image	31
17	java.text	21	35	javax.swing.border	31
18	java.awt.event	22	36	junit.framework	31

6.2 Business & Enterprise category

We found 35 relevant Java software in Sourcerer dataset corresponding exclusively to Business & Enterprise SourceForge category. In Table 13, we showed those software and their number of APIs.

Table 13 Baseline data for Business & Enterprise category

#	SOFTWARE NAME	NUMBER OF APIs
1	MoneyJar	9
2	LunchTime	9
3	timetowork	9
4	ibtools.ibcontroller	10
5	GnuCashToQIF	12
6	realm	12
7	saCASH	12
8	Installer	14
9	chameleon	16
10	HDS	16
11	POM	17
12	MOSES2	18
13	BlackSheepProjekt	19
14	ToDo	19
15	sfljTSE	19
16	convelo-base	20
17	OpenAdmin	20
18	CleanSheets	23
19	doyen	26
20	Parking Garage	27
21	epoline_jsf_service_Dossier	29
22	Settle	32
23	tag control	34
24	eLawManager	40
25	osoptiek_sf	42
26	timeslottracker	42
27	BookKeeper	48
28	inidonaTimeTracker	52
29	pandora	53
30	DDC	56
31	jproject	56
32	ganttproject	87
33	JGnash2	106
34	processdash	110
35	ofbizNeogia	181

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Business & Enterprise category regardless of the stage of the software development, we did the 5-fold cross validation (Figure 21). Hence, in every iteration we used seven different software (i.e., 20% of data) as test set and 28 software (i.e., 80% of data) as training set. In that partitioning, we avoided overlapping, i.e., every software appears just once in test set. In this partitioning process, we saved `.wst` files of every iteration.

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our

recommendation methodology in every stage of API recommendations. We exposed these results as follows.

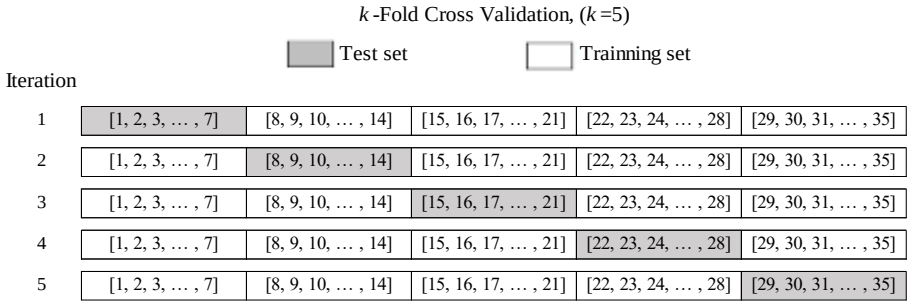

Figure 21 Business & Enterprise category dataset partition

6.2.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 14). From 0.1 to 0.4, *minSupport* presented same results with the highest recall. Hence, we chose *minSupport* = 0.4 as the "best" *minSupport value* for FIS technique in Business & Enterprise category, since *minSupport* = 0.4 offered a little improvement in precision value.

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the Business & Enterprise category and in Stage A of development when receiving large lists of APIs recommended as well as when receiving top-*N* list of

APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Table 14 Effect of varying *minSupport* value in top-10 lists of APIs recommended in Stage A of Business & Enterprise category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.258	0.623	1.000
0.2	0.258	0.623	1.000
0.3	0.258	0.623	1.000
0.4	0.258	0.631	1.000
0.5	0.242	0.641	1.000
0.6	0.165	0.675	1.000
0.7	0.098	0.876	1.000
0.8	0.093	0.986	1.000
0.9	0.093	0.986	1.000

Regarding the evaluation when receiving lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Table 15). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 22.

Table 15 Results of evaluation metrics for large lists of APIs recommended in Stage A of Business & Enterprise category.

recall	**precision**	**recall rate**
0.278	0.595	1.000

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 16). In addition, for analyzing every iteration, we exposed the results of precision (Figure 23), recall (Figure 24), and recall rate (Figure 25) for each iteration.

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. In Table 17, we showed number of recommended

APIs, their name and their frequency value, where frequency is the number of times the API was recommended over the five iterations.

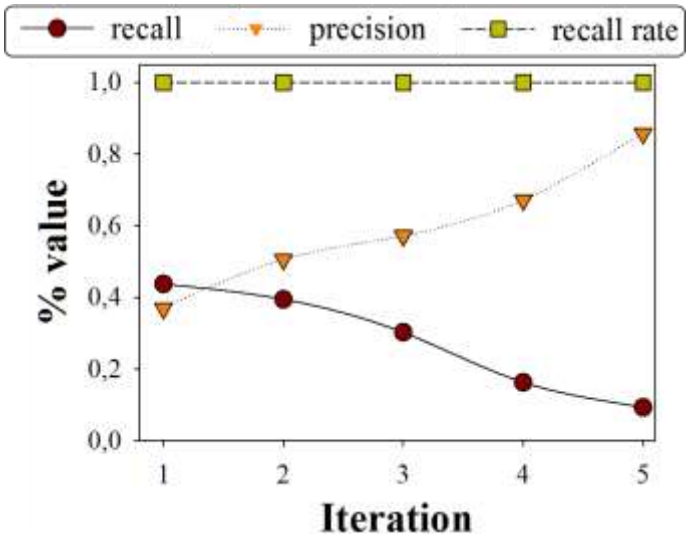

Figure 22 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of Business & Enterprise category.

Table 16 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of Business & Enterprise category.

N	**recall**	**precision**	**recall rate**
1	0.048	1.000	1.000
3	0.107	0.800	1.000
5	0.154	0.703	1.000
7	0.203	0.686	1.000
10	0.258	0.631	1.000
13	0.278	0.595	1.000
15	0.278	0.595	1.000
17	0.278	0.595	1.000
20	0.278	0.595	1.000

6.2.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

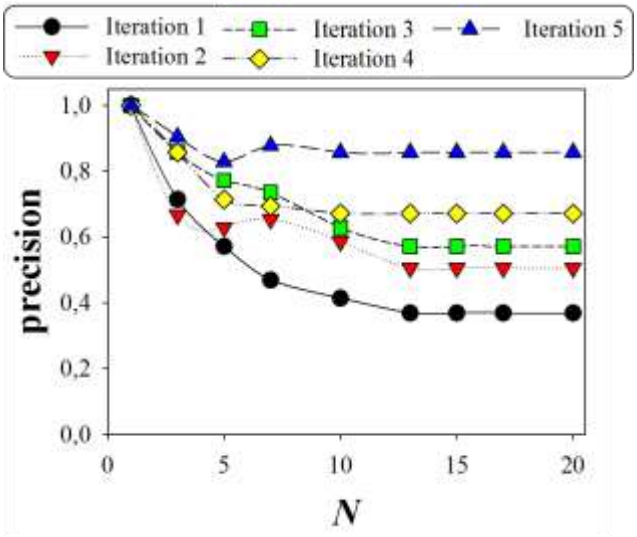

Figure 23 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Business & Enterprise category.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.4 found for FIS technique in Stage A, since we used the same data

sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 18) and we found that varying *k* in 15, 20, and 25 presented the same best recall values. Thus, we arbitrarily chose *k* = 25, since choosing other did not mean any improvement, either for precision or recall rate values. As a result, we chose *minSupport* = 0.4 for FIS technique and *k* = 25 for CF technique in Business & Enterprise category.

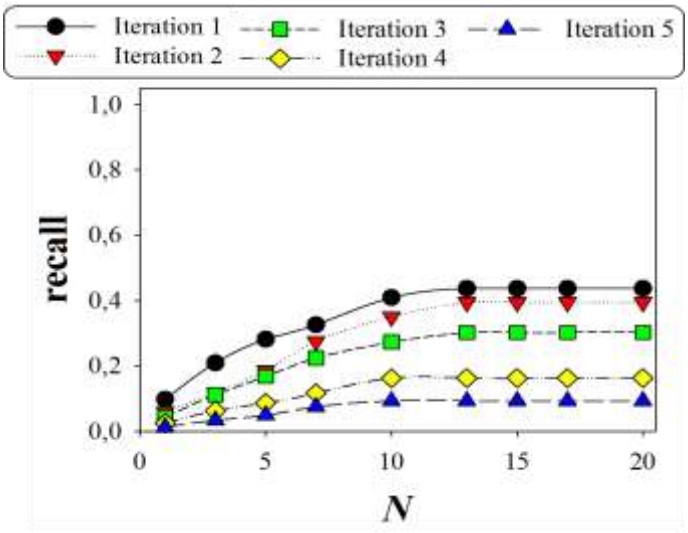

Figure 24 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Business & Enterprise category.

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in Business & Enterprise category and Stage B of development when receiving large lists of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In the case of top-*N*, we evaluated the effect of varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving large lists of recommended APIs, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations and replicates of the 5-fold cross validation (Table 19). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 26.

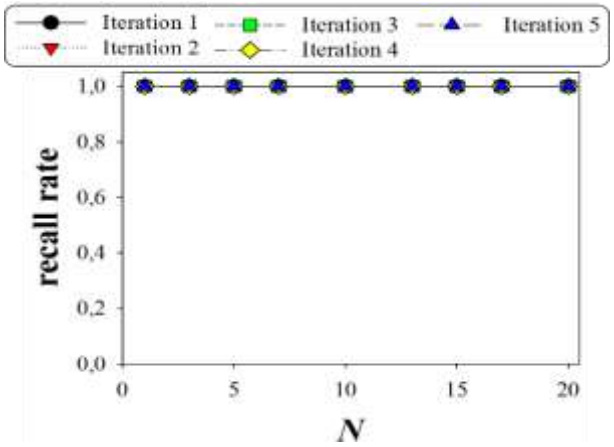

Figure 25 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Business & Enterprise category.

Table 17 APIs recommended in top-20 lists in Stage A of Business & Enterprise category

#	API	Frequency
1	javax.swing.event	1
2	java.util.regex	1
3	org.xml.sax	2
4	java.sql	3
5	javax.xml.parsers	4
6	java.lang.reflect	5
7	java.util	5
8	java.awt	5
9	java.text	5
10	java.net	5
11	java.io	5
12	java.awt.event	5
13	javax.swing.border	5
14	javax.swing	5

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 20). In addition, for analyzing every iteration, we expose average results of precision (Figure 27), recall (Figure 28), and recall rate (Figure 29) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly (Section 5.4).

Table 18 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of Business & Enterprise category.

k	**recall**	**precision**	**recall rate**
5	0.264	0.528	0.966
10	0.267	0.500	0.966
15	0.268	0.500	0.966
20	0.268	0.500	0.966
25	0.268	0.500	0.966

Table 19 Results of evaluation metrics for large lists of APIs recommended in Stage B of Business & Enterprise category.

recall	**precision**	**recall rate**
0.256	0.502	0.971

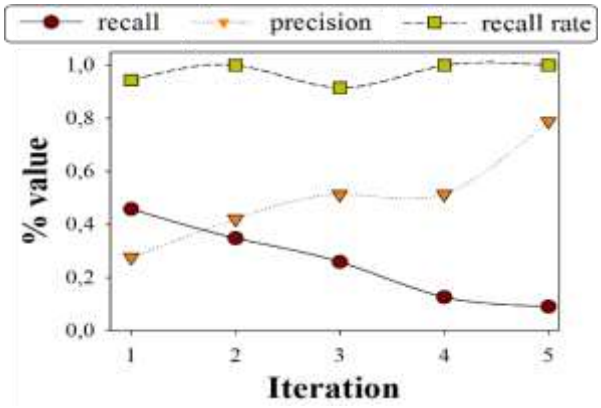

Figure 26 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of Business & Enterprise category.

Table 20 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of Business & Enterprise category.

N	recall	precision	recall rate
1	0.079	0.874	0.874
3	0.159	0.629	0.920
5	0.209	0.548	0.960
7	0.245	0.519	0.966
10	0.254	0.502	0.971
13	0.256	0.502	0.971
15	0.256	0.502	0.971
17	0.256	0.502	0.971
20	0.256	0.502	0.971

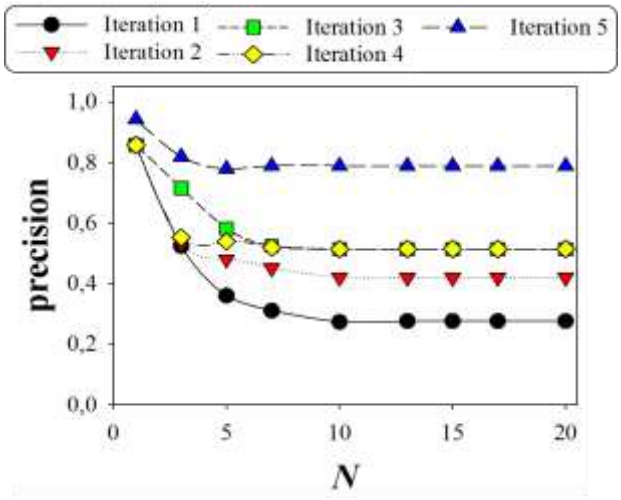

Figure 27 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Business & Enterprise category.

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. As in Stage B of API recommendation we made five replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value. In this category, we found that replicates 1, 2, 2, 1, and 5 presented the best recall values for iterations 1, 2, 3, 4, and 5 respectively. In Table 21, we showed the number of APIs

recommended, the APIs, and frequency value, where frequency is the number of times the API was recommended in a top-20 list lists over the 35 target software.

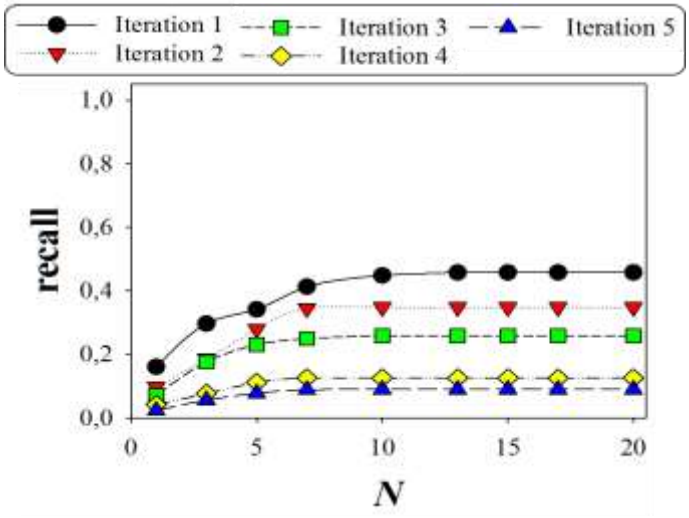

Figure 28 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Business & Enterprise category.

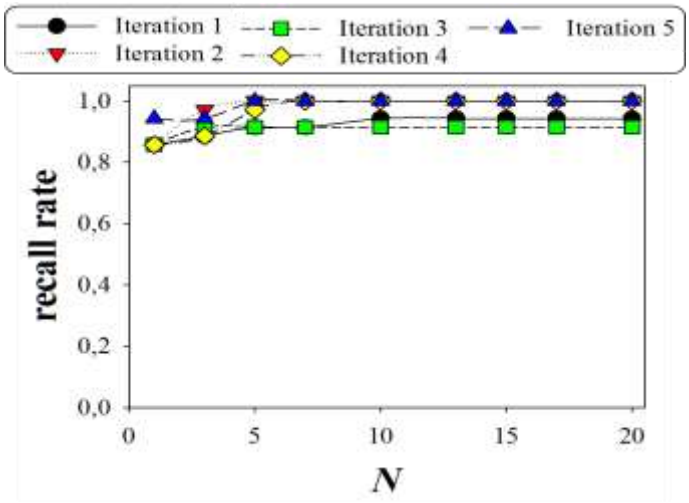

Figure 29 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Business & Enterprise category.

Table 21 APIs recommended in top-20 lists in Stage B of Business & Enterprise category.

#	API	Frequency
1	javax.swing	15
2	javax.xml.parsers	16
3	javax.swing.border	17
4	java.util	18
5	java.io	19
6	java.sql	21
7	java.awt	25
8	java.net	28
9	java.lang.reflect	29
10	java.text	30
11	java.awt.event	31

6.3 Communications category

We found 50 relevant Java software in Sourcerer dataset corresponding exclusively to Communications SourceForge category. In Table 22, we showed those software and their number of APIs.

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Communications category regardless of the stage of software development, we did the 5-fold cross validation (Figure 30). Hence, in every iteration, we used 10 different software (i.e., 20% of data) as test set and 40 software (i.e., 80% of data) as training set. In that partitioning, we avoided overlapping, i.e., every software appears just once in the test set. In this partitioning process, we saved `.wst` files of every iteration.

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation methodology in every stage of API recommendations. We exposed these results as follows.

Table 22 Baseline data for Communications category

#	SOFTWARE NAME	NUMBER OF APIs
1	JOpenPhone	7
2	tarproxy	8
3	sendmail-jilter	8
4	gnu-hylafax	9
5	irclib	9
6	Jim	9
7	netmessenger	9
8	rsswaba	10
9	OWL	10
10	MezlGmail	12
11	james-ha	13
12	jisdncall	13
13	PFC_ClienteMail_v6	14
14	PastePro - Release	14
15	DESK_MESSAGE	15
16	Jasim	15
17	smsar	15
18	BinPost	16
19	jmailserverJML	16
20	ProperJavaRDP	16
21	REOAnalyzer	16
22	Petridish Chatter	17
23	POP Surgeon	17
24	cwterm	18
25	Calls	18

#	SOFTWARE NAME	NUMBER OF APIs
26	furthurnet	18
27	Asterisk	18
28	PJIRC	18
29	jalk	20
30	JOscarLib	21
31	linuxwifi	21
32	together-jsmime	21
33	spambuster	21
34	ararat	24
35	Web-portal	25
36	phoobot	26
37	Bittorrent	27
38	mflow	27
39	iChat LE v2	30
40	JML	30
41	JavaDC	34
42	groupscheme	42
43	zim	43
44	xBus	45
45	jhylafax	47
46	SoftPhone	52
47	jfritz	54
48	spamato	56
49	fr-wot	57
50	limewire_spam	65

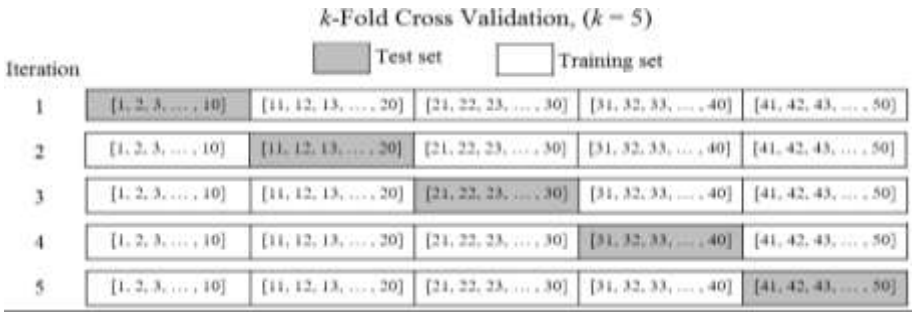

Figure 30 Communications category dataset partition

6.3.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 23). From 0.1 to 0.3, *minSupport* presented same results with the highest recall. Hence, we chose *minSupport* = 0.1 as the "best" *minSupport value* for FIS technique in Communications category, since choosing other *minSupport* value did not mean any significant improvement either for precision or recall rate values.

Table 23 Effect of varying *minSupport* value technique in top-10 lists of APIs recommended in Stage A of Communications category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.330	0.626	0.980
0.2	0.330	0.626	0.980
0.3	0.330	0.626	0.980
0.4	0.314	0.662	0.980
0.5	0.280	0.753	0.980
0.6	0.229	0.802	0.980
0.7	0.178	0.884	0.980
0.8	0.158	0.940	0.960
0.9	0.135	0.940	0.960

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the Audio & Video category and in Stage A of development when receiving

large lists of APIs recommended as well as when receiving top-*N* list of APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Table 24). In addition, for analyzing every iteration, we exposed the evaluation results for each one in Figure 31.

Table 24 Results of evaluation metrics for large lists of APIs recommended in Stage A of Communications category.

recall	precision	recall rate
0.582	0.264	0.980

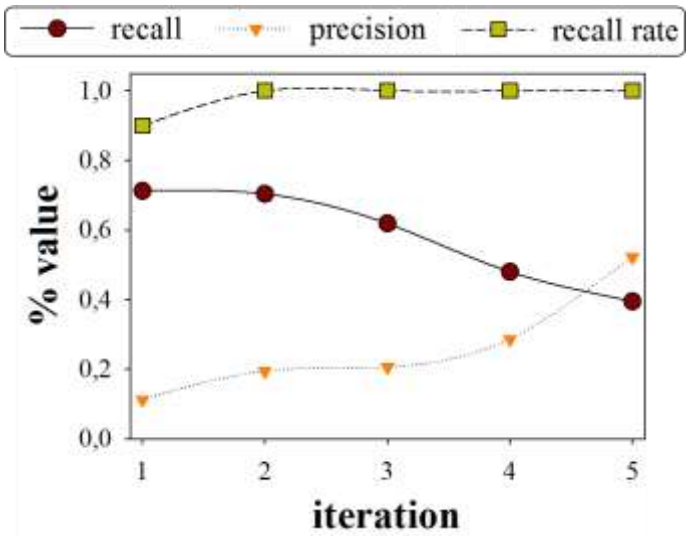

Figure 31 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of Communications category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 25). In addition, for analyzing

every iteration, we exposed the results of precision (Figure 32), recall (Figure 33), and recall rate (Figure 34) for each iteration.

After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. In Table 26, we showed number of recommended APIs, their name, and the corresponding frequency value, where frequency is the number of times the API was recommended over the five iterations.

Table 25 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of Communications category.

N	**recall**	**precision**	**recall rate**
1	0.055	0.960	0.960
3	0.158	0.940	0.960
5	0.226	0.820	0.980
7	0.283	0.749	0.980
10	0.330	0.626	0.980
13	0.368	0.546	0.980
15	0.383	0.495	0.980
17	0.396	0.459	0.980
20	0.412	0.413	0.980

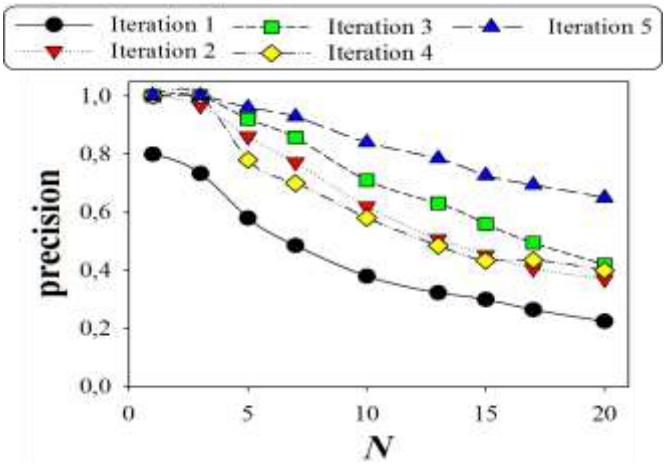

Figure 32 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Communications category.

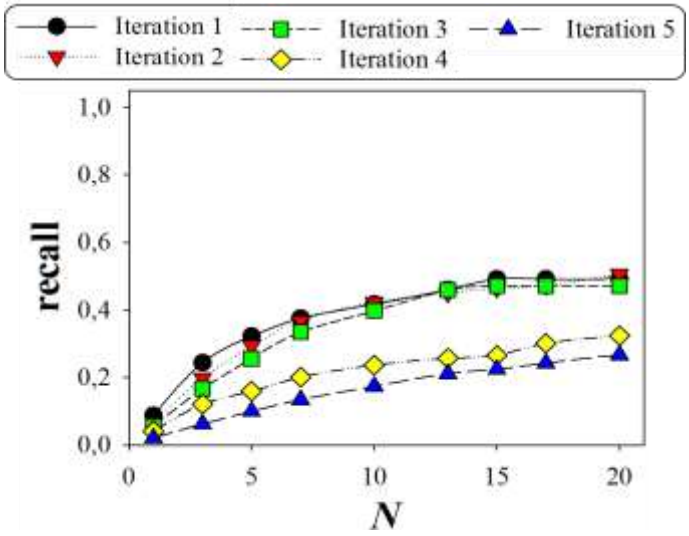

Figure 33 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Communications category.

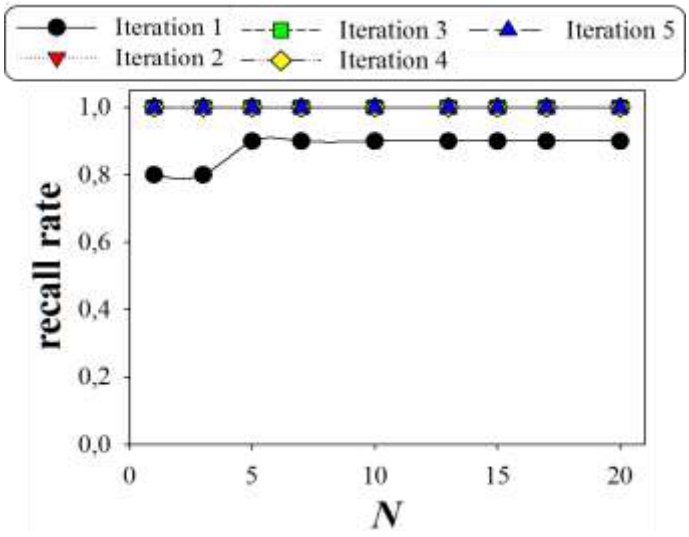

Figure 34 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Communications category.

6.3.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

Table 26 APIs recommended in top-20 lists in Stage A of Communications category

#	API	Frequency	#	API	Frequency
1	java.util.zip	1	15	java.util	5
2	java.util.regex	1	16	javax.swing.table	5
3	org.apache.log4j	1	17	java.net	5
4	java.util.logging	2	18	org.xml.sax	5
5	javax.mail.internet	2	19	java.lang.reflect	5
6	javax.mail	2	20	java.security	5
7	javax.xml.parsers	2	21	javax.swing.event	5
8	javax.swing.tree	2	22	java.awt	5
9	org.apache.commons.logging	3	23	java.text	5
10	javax.swing.border	3	24	java.io	5
11	java.awt.image	4	25	java.awt.event	5
12	java.beans	4	26	junit.framework	5
13	org.w3c.dom	4	27	javax.swing	5
14	javax.swing.text	4			

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found for FIS technique in Stage A, since used the same data sample, technique, and evaluation method. On the other hand, we used the plug-in and

analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 27) and we found that varying *k* in 5 presented the best recall and precision values. Consequently, we chose *minSupport* = 0.10 for FIS technique and *k* = 5 for CF technique in Communications category,

Table 27 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of Communications category.

k	**recall**	**precision**	**recall rate**
5	0.369	0.380	0.972
10	0.357	0.365	0.972
15	0.360	0.370	0.968
20	0.364	0.375	0.968
25	0.363	0.372	0.968

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in Communications category and stage B of development large lists of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In case of top-*N*, we evaluated the effect of varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations and replicates of the 5-fold cross validation (Table 28). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 35.

Table 28 Results of evaluation metrics for large lists of APIs recommended in Stage B of Communications category.

recall	**precision**	**recall rate**
0.482	0.263	0.980

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from

all of iterations of the 5-fold cross validation (Table 29). In addition, for analyzing every iteration, we expose average results of precision (Figure 36), recall (Figure 37), and recall rate (Figure 38) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly (Section 5.4).

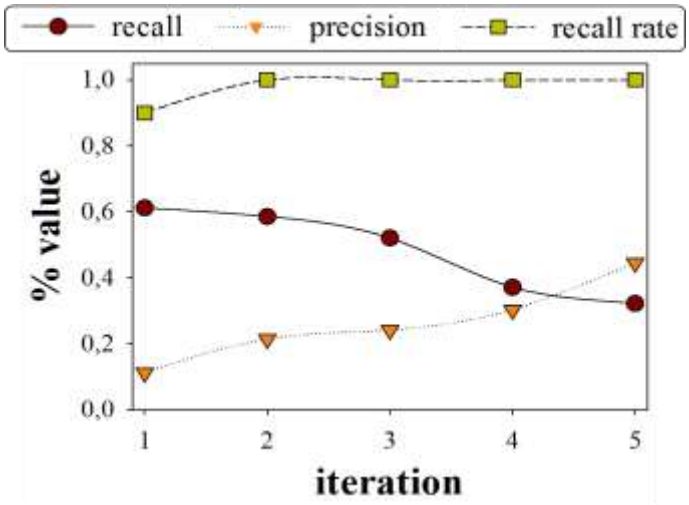

Figure 35 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of Communications category.

Table 29 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of Communications category.

N	**recall**	**precision**	**recall rate**
1	0.101	0.928	0.928
3	0.247	0.776	0.940
5	0.320	0.618	0.964
7	0.365	0.515	0.972
10	0.405	0.410	0.976
13	0.427	0.343	0.976
15	0.441	0.313	0.976
17	0.453	0.293	0.976
20	0.466	0.276	0.976

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. As in Stage B of API recommendation we made five

replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value. In this category, we found that replicates 3, 4, 1, 5, and 1 presented the best recall values for iterations 1, 2, 3, 4, and 5 respectively. In Table 30, we showed the number of APIs recommended, the APIs, and frequency value, where frequency is the number of times the API was recommended in a top-20 list over the 50 target software.

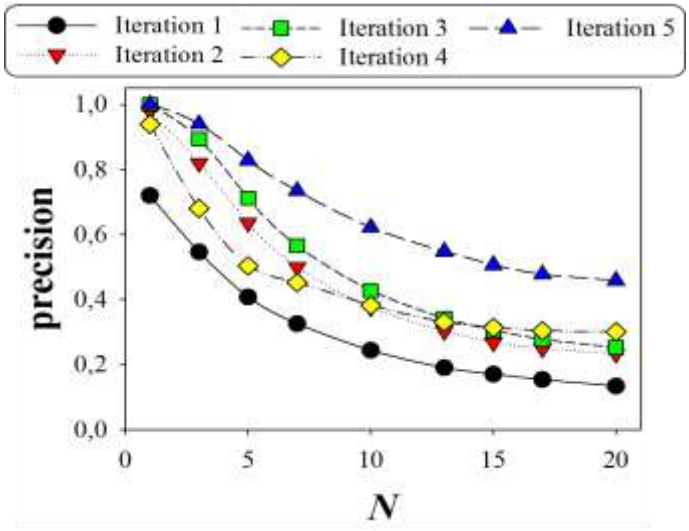

Figure 36 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Communications category.

6.4 Games category

We found 70 relevant Java software in the Sourcerer dataset corresponding exclusively to Games SourceForge category. In Table 31, we showed those software and their number of APIs.

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Games category regardless of the stage of software development, we did the 5-fold cross validation (Figure 39). Hence, in every iteration we used 14 different software (i.e., 20% of data) as test set and 56 software (i.e., 80% of data) as training set. In that partitioning, we avoided

overlapping, i.e., every software appears just once in test set. In this partitioning process, we saved `.wst` files of every iteration.

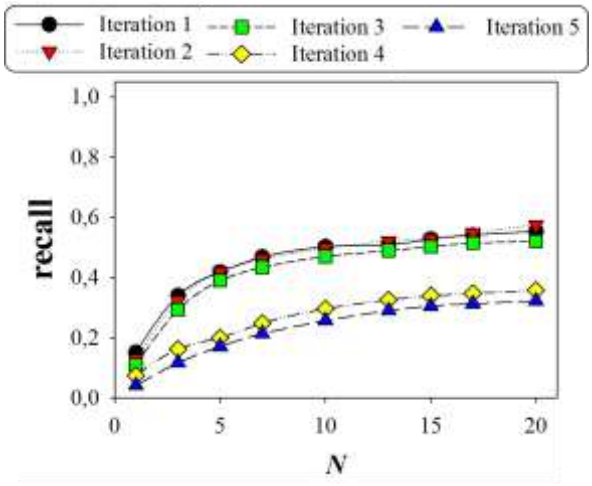

Figure 37 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Communications category.

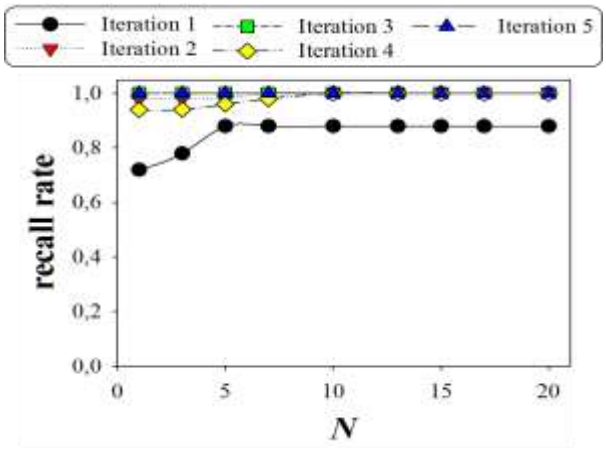

Figure 38 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Communications category.

Table 30 APIs recommended in top-20 lists in Stage B of Communications category

#	API	Frequency
1	javax.swing.plaf.basic	2
2	javax.swing.filechooser	3
3	java.security.spec	5
4	org.apache.xerces.parsers	5
5	java.security.cert	6
6	org.xml.sax.helpers	6
7	javax.crypto.spec	6
8	javax.crypto	7
9	javax.swing.plaf.metal	7
10	org.apache.commons.logging	9
11	javax.activation	11
12	java.nio.channels	12
13	javax.net.ssl	12
14	java.util.logging	14
15	java.util.regex	14
16	javax.mail.internet	15
17	javax.sound.sampled	15
18	java.sql	18
19	java.util.zip	18
20	javax.swing	18
21	java.nio	19
22	java.math	20
23	javax.mail	22

#	API	Frequency
24	org.w3c.dom	23
25	java.applet	24
26	javax.xml.parsers	24
27	java.awt.datatransfer	24
28	org.apache.log4j	24
29	java.awt.image	25
30	javax.swing.tree	25
31	org.xml.sax	26
32	junit.framework	27
33	java.util	31
34	java.beans	31
35	java.net	31
36	javax.swing.table	32
37	javax.swing.text	32
38	java.lang.reflect	33
39	java.awt.event	34
40	java.security	36
41	java.text	37
42	java.io	37
43	javax.swing.border	38
44	javax.swing.event	40
45	java.awt	40

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation methodology in every stage of API recommendations. We exposed these results as follows.

6.4.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed

and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

Table 31 Baseline data for Games category

#	SOFTWARE NAME	NUMBER OF APIs	#	SOFTWARE NAME	NUMBER OF APIs
1	Sudoku	5	36	TwinSerpents	17
2	go2	6	37	Xoridor-SF	18
3	lightsout	6	38	Tiffanys	18
4	jSweeper	6	39	holdemcockpit	19
5	Vana'diel Timer	6	40	jcharmanager	19
6	tetris	6	41	robowars	19
7	CarolSolitaire	7	42	thud-1.4	19
8	mkchess	7	43	bobeira	20
9	JHex	7	44	bzstats	20
10	openjmud	7	45	openkickoff	20
11	Vorms	7	46	Olitext	20
12	EnergyBolt	8	47	hogs	21
13	problematic	8	48	PJShadowsFall	21
14	NebulaCards	8	49	Bomberman	22
15	talisman	9	50	JTBRPG	22
16	conwaygo	10	51	DarkWorld	23
17	Blasteroids	10	52	Ice Hockey Manager	24
18	minesweeper	10	53	Errare	24
19	Chat	11	54	JMines	24
20	momem	11	55	JurpeEclipse	25
21	customsrpg	12	56	rcontrol	25
22	freya-working	12	57	Herzog3D	26
23	Jacoto	12	58	puzzlebeans	26
24	JOBS	12	59	SpaceWars	28
25	JEdits	13	60	jake2	29
26	tweevoortwaalf	13	61	kolmafia	30
27	go-3	14	62	rpg-mapgen	31
28	PirateMoon	14	63	bertelConf	33
29	dragonchess	15	64	mulifex	34
30	JDStar	15	65	universe	35
31	ROOT	15	66	Magellan	36
32	Zatacka Online	15	67	au.com.kelpie.rcpplanner	50
33	battlephone	16	68	de.battleforge	51
34	TaroTux	16	69	RouteRuler	58
35	stonesthrow	16	70	ho_plugins	108

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the “best” threshold value, i.e., the “best” *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS

technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 32). The *minSupport* values in 0.1 and 0.2 presented same results with the highest recall. Hence, we chose *minSupport* = 0.1 as the "best" *minSupport value* for FIS technique in Games category, since choosing other *minSupport* value did not mean any significant improvement either for precision or recall rate values.

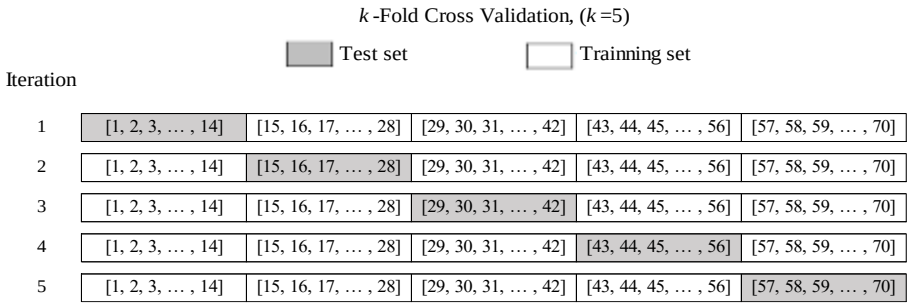

Figure 39 Games category dataset partition

Table 32 Effect of varying *minSupport* value in top-10 lists of APIs recommended in Stage A of Games category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.459	0.686	1.000
0.2	0.459	0.686	1.000
0.3	0.455	0.688	1.000
0.4	0.441	0.718	1.000
0.5	0.407	0.782	1.000
0.6	0.368	0.832	1.000
0.7	0.336	0.878	1.000
0.8	0.322	0.889	1.000
0.9	0.100	0.929	1.000

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the Games category and in Stage A of development when receiving large lists of APIs recommended as well as when receiving top-*N* list of APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Table 33). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 40.

Table 33 Results of evaluation metrics for large lists of APIs recommended in Stage A of Games category.

recall	precision	recall rate
0.660	0.337	1.000

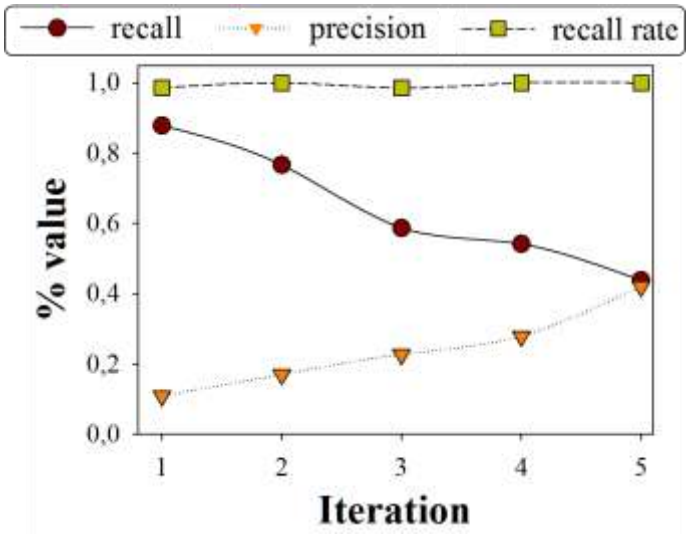

Figure 40 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of Games category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 34). In addition, for analyzing every iteration, we exposed the results of precision (Figure 41), recall (Figure 42), and recall rate (Figure 43) for each iteration.

After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. In Table 35, we showed number of recommended

APIs, their name and their frequency value, where frequency is the number of times the API was recommended over the five iterations.

Table 34 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of Games category.

N	recall	precision	recall rate
1	0.073	0.986	0.986
3	0.198	0.924	1.000
5	0.322	0.889	1.000
7	0.388	0.794	1.000
10	0.459	0.686	1.000
13	0.514	0.608	1.000
15	0.533	0.550	1.000
17	0.550	0.506	1.000
20	0.573	0.454	1.000

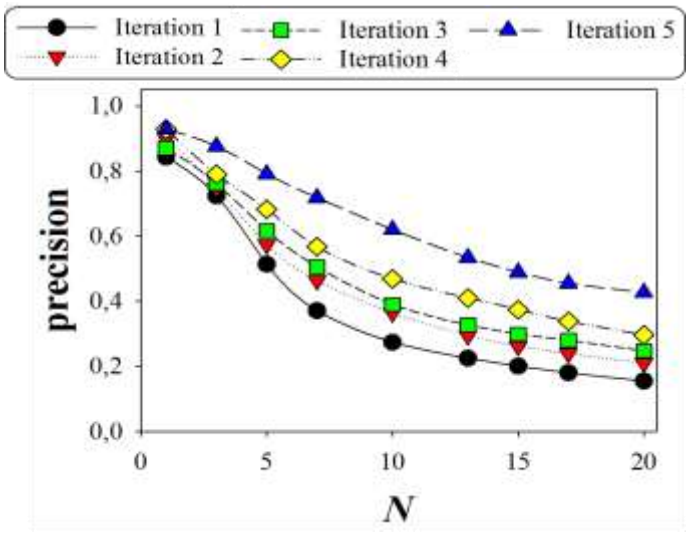

Figure 41 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Games category.

6.4.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

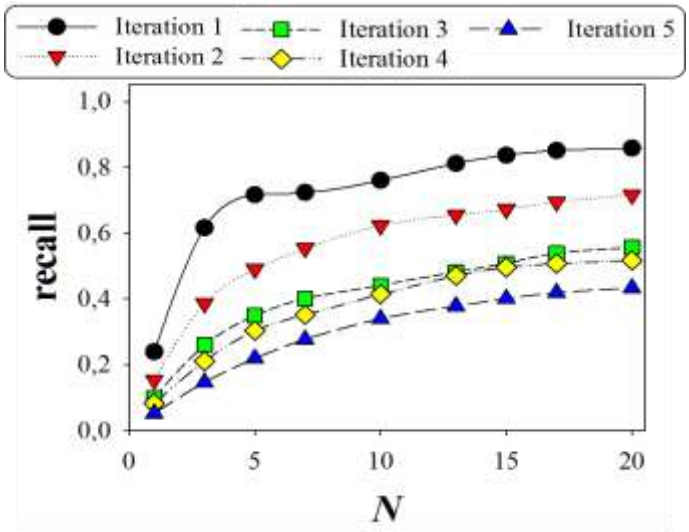

Figure 42 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Games category.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found for FIS technique in Stage A, since we used the same data

sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 36) and we found that varying *k* in 20 presented the best recall value. Consequently, we chose *minSupport* = 0.1 for FIS technique and *k* = 20 for CF technique in Games category.

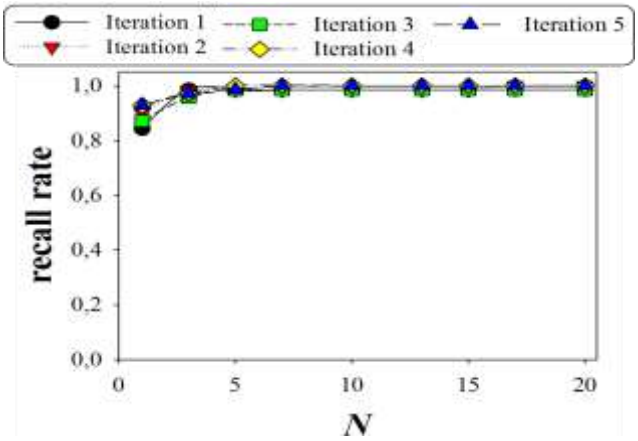

Figure 43 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Games category.

Table 35 APIs recommended in top-20 lists in Stage A of Games category

#	API	Frequency
1	org.jdom	1
2	java.nio	1
3	javax.sound.sampled	1
4	java.util.logging	2
5	java.beans	2
6	javax.swing.text	3
7	javax.xml.parsers	3
8	java.util.zip	4
9	org.xml.sax	4
10	javax.swing.filechooser	4
11	java.util	5
12	java.awt.image	5
13	javax.swing.table	5
14	java.net	5
15	java.lang.reflect	5
16	java.awt.geom	5
17	javax.imageio	5
18	javax.swing.event	5
19	java.awt	5
20	java.text	5
21	java.io	5
22	java.awt.event	5
23	javax.swing.border	5
24	junit.framework	5
25	javax.swing	5

Table 36 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of Games category.

k	**recall**	**precision**	**recall rate**
5	0.494	0.453	0.989
10	0.509	0.411	0.989
15	0.513	0.415	0.989
20	0.518	0.419	0.989
25	0.512	0.416	0.989

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in Games category and Stage B of development when receiving lists of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In case of top-*N*, we evaluated the effect of varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations and replicates of the 5-fold cross validation (Table 37). In addition, for analyzing every iteration, we exposed evaluation results in Figure 44.

Table 37 Results of evaluation metrics for large lists of APIs recommended in Stage B of Games category.

recall	**precision**	**recall rate**
0.643	0.241	0.994

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 38). In addition, for analyzing every iteration, we expose average results of precision (Figure 45), recall (Figure 46), and recall rate (Figure 47) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly (Section 5.4).

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. As in Stage B of API recommendation we made five replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value. In this category, we found that replicates 3, 4, 5, 2, and 2 presented the best recall values for iterations 1, 2, 3, 4, and 5 respectively. In Table 39, we showed the number of APIs recommended, the APIs, and frequency value, where frequency is the number of times the API was recommended in a top-20 list over the 70 target software.

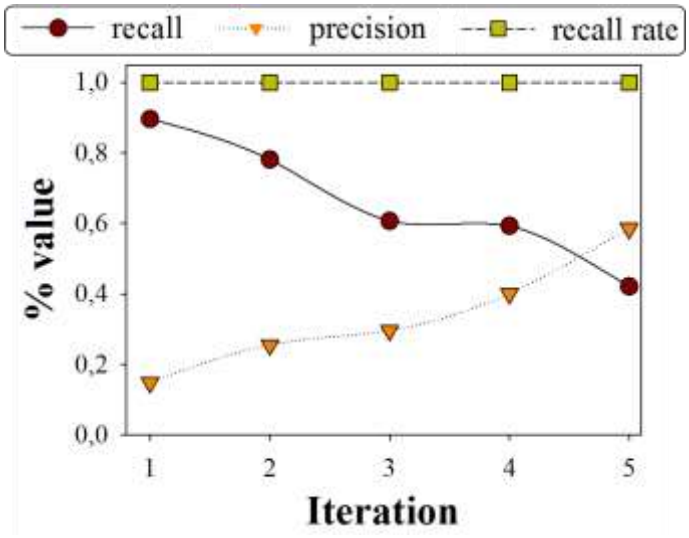

Figure 44 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of Games category.

6.5 Graphics category

We found 20 relevant Java software in the Sourcerer dataset corresponding exclusively to Graphics SourceForge category. In Table 40, we showed those software and their number of APIs.

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Graphics category r regardless of the stage of software development, we did the 5-fold cross validation (Figure 48). Hence,

in every iteration we used four different software (i.e., 20% of data) as test set and 16 software (i.e., 80% of data) as training set. In that partitioning, we avoided overlapping, i.e., every software appears just once in the test set. In this partitioning process, we saved `.wst` files of every iteration.

Table 38 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of Games category.

N	**recall**	**precision**	**recall rate**
1	0.125	0.894	0.894
3	0.324	0.780	0.974
5	0.416	0.636	0.989
7	0.462	0.525	0.991
10	0.516	0.423	0.991
13	0.560	0.358	0.991
15	0.583	0.325	0.991
17	0.602	0.299	0.994
20	0.616	0.267	0.994

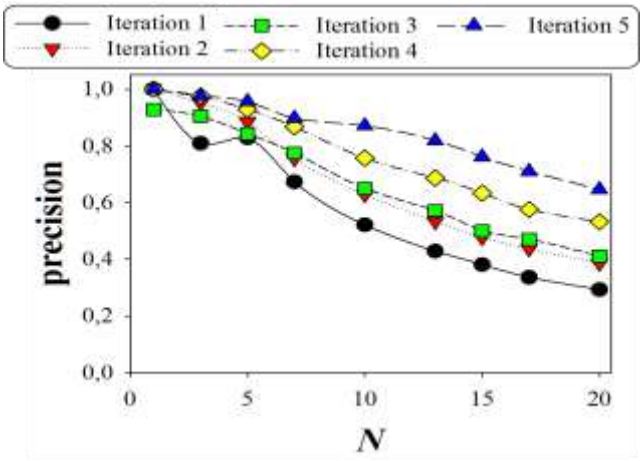

Figure 45 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Games category.

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied

the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation methodology in every stage of API recommendations. We exposed these results as follows.

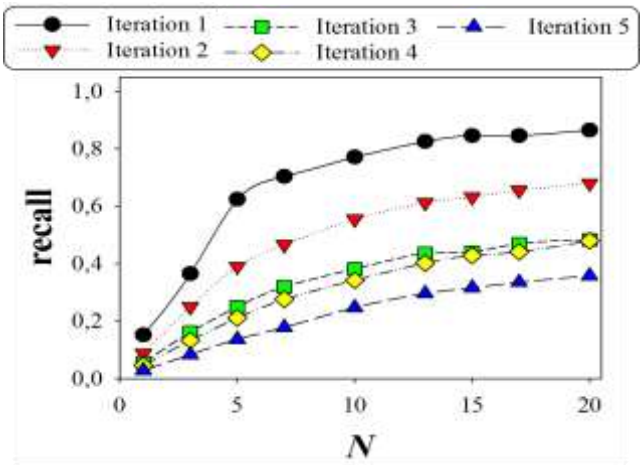

Figure 46 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Games category.

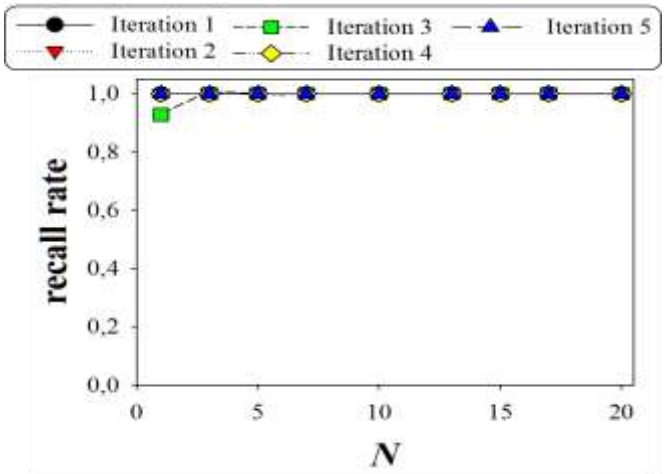

Figure 47 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Games category.

Table 39 APIs recommended in top-20 lists in Stage B of Games category

#	API	Frequency	#	API	Frequency
1	org.apache.log4j	1	18	java.util	46
2	java.awt.font	5	19	java.beans	47
3	java.security	5	20	java.net	52
4	java.sql	9	21	javax.imageio	52
5	java.util.regex	9	22	javax.swing.event	53
6	javax.swing	15	23	javax.swing.text	54
7	java.nio	21	24	javax.swing.border	54
8	org.jdom	22	25	java.awt.image	55
9	java.util.logging	33	26	org.xml.sax	55
10	java.applet	34	27	javax.xml.parsers	57
11	org.jdom.input	36	28	javax.sound.sampled	58
12	javax.swing.tree	36	29	java.lang.reflect	59
13	org.w3c.dom	37	30	java.awt.geom	60
14	java.io	41	31	java.text	60
15	java.awt.event	41	32	javax.swing.filechooser	61
16	java.awt	43	33	junit.framework	61
17	java.util.zip	45	34	javax.swing.table	66

Table 40 Baseline data for Graphics category

#	SOFTWARE NAME	NUMBER OF APIs	#	SOFTWARE NAME	NUMBER OF APIs
1	jcam	8	11	jargs-modified	18
2	jiu	9	12	sf.net GooRaph	19
3	lsketch	11	13	Iupiter2	21
4	AccolorHelper	12	14	particleRealityCVS	21
5	Subdivision	12	15	Lenticular	22
6	JPhotoTweek	12	16	JVector	24
7	Java Image CD-Rom	15	17	drafts	25
8	CDPhotoIndex	16	18	JPatch	25
9	oppoc	18	19	sunflow	29
10	Xmall_0.2.2	18	20	jnetvis	42

6.5.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

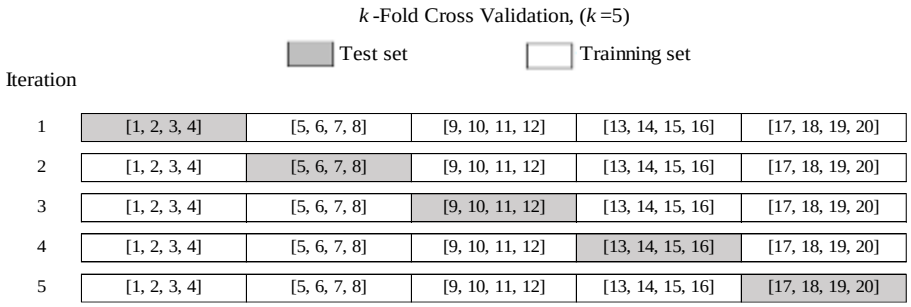

Figure 48 Graphics category dataset partition

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 41). From 0.1 to 0.3, *minSupport* presented same results with the highest recall. Hence, we chose *minSupport* = 0.1 as the "best" *minSupport value* for FIS technique in Graphics category, since choosing other *minSupport* value did not mean any significant improvement either for precision or recall rate values.

Table 41 Effect of varying *minSupport* value in top-10 lists of APIs recommended in in Stage A of Graphics category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.424	0.725	1.000
0.2	0.424	0.725	1.000
0.3	0.424	0.725	1.000
0.4	0.418	0.729	1.000
0.5	0.409	0.740	1.000
0.6	0.364	0.810	1.000
0.7	0.308	0.856	1.000
0.8	0.264	0.908	1.000
0.9	0.143	0.967	1.000

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the Graphics category and in Stage A of development when receiving large lists

of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Table 42). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 49.

Table 42 Results of evaluation metrics for large lists of APIs recommended in Stage A of Graphics category.

recall	precision	recall rate
0.654	0.260	1.000

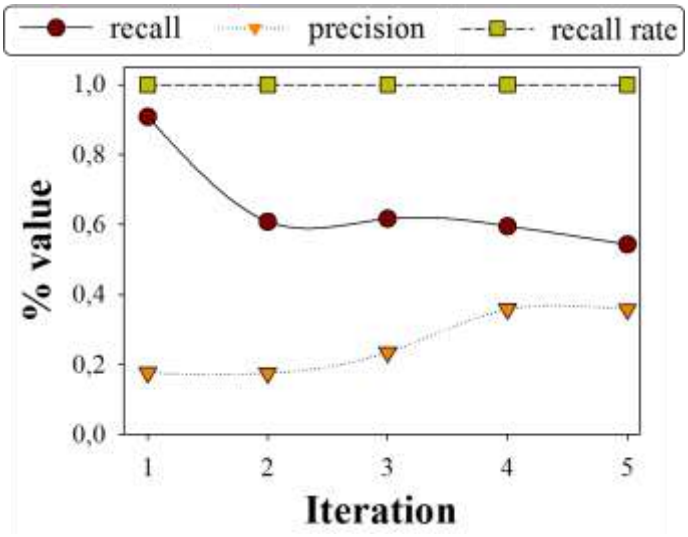

Figure 49 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of Graphics category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 43). In addition, for analyzing

every iteration, we exposed the results of precision (Figure 50), recall (Figure 51), and recall rate (Figure 52) for each iteration.

Table 43 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of Graphics category.

N	**recall**	**precision**	**recall rate**
1	0.062	1.000	1.000
3	0.178	0.950	1.000
5	0.276	0.890	1.000
7	0.351	0.829	1.000
10	0.424	0.725	1.000
13	0.498	0.669	1.000
15	0.505	0.583	1.000
17	0.510	0.524	1.000
20	0.527	0.463	1.000

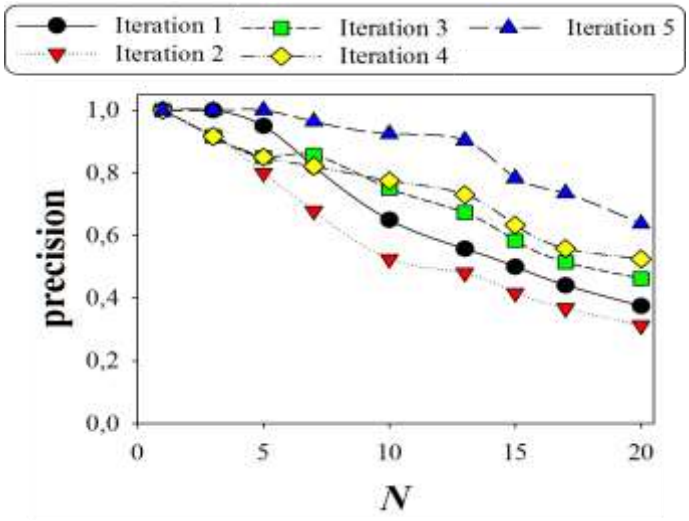

Figure 50 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Graphics category.

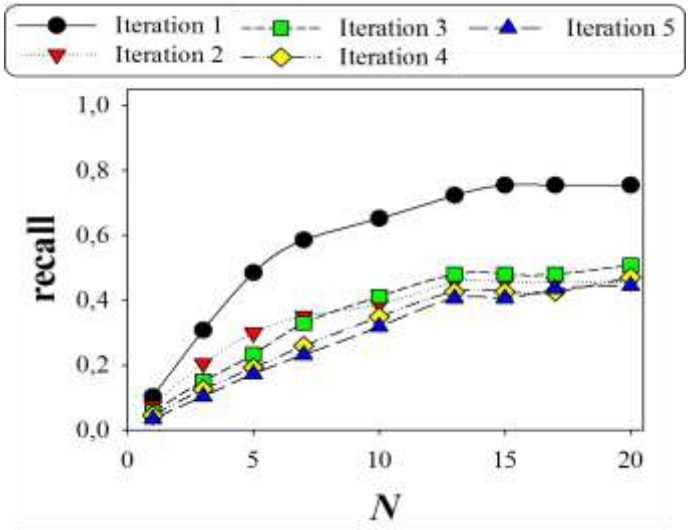

Figure 51 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Graphics category.

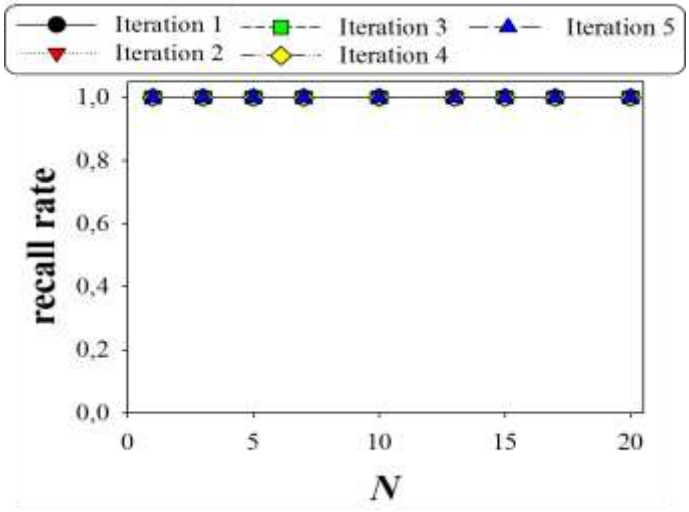

Figure 52 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Graphics category.

After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. In Table 44, we showed number of recommended

APIs, their name and their frequency value, where frequency is the number of times the API was recommended over the five iterations.

Table 44 APIs recommended in top-20 lists in Stage A of Graphics category

#	API	Frequency
1	org.xml.sax	1
2	java.lang.reflect	1
3	javax.swing.text.html	1
4	javax.swing.tree	1
5	java.applet	2
6	java.util.zip	2
7	javax.imageio.stream	2
8	javax.vecmath	2
9	java.util.regex	3
10	java.awt.datatransfer	3
11	java.beans	4
12	java.awt.font	4
13	javax.swing.text	4
14	java.util	5
15	java.awt.image	5
16	java.nio	5
17	java.net	5
18	javax.swing.filechooser	5
19	java.awt.geom	5
20	javax.imageio	5
21	javax.swing.event	5
22	java.awt	5
23	java.text	5
24	java.io	5
25	java.awt.event	5
26	javax.swing.border	5
27	javax.swing	5

6.5.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found for FIS technique in Stage A, since we used the same data

sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 45) and we found that varying *k* in 15 presented the best recall value. Consequently, we chose *minSupport* = 0.1 for FIS technique and *k* = 15 for CF in Graphics category,

Table 45 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of Graphics category.

k	**recall**	**precision**	**recall rate**
5	0.486	0.436	0.442
10	0.486	0.439	0.444
15	0.487	0.437	0.443

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in Graphics category and Stage B of development when receiving large lists of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations and replicates of the 5-fold cross validation (Table 46). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 53.

Table 46 Results of evaluation metrics for large lists of APIs recommended in Stage B of Graphics category.

recall	**precision**	**recall rate**
0.667	0.150	1.000

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 47). In addition, for analyzing every iteration, we expose average results of precision (Figure 54), recall (Figure 55), and recall rate (Figure 56) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly (Section 5.4).

Table 47 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of Graphics category.

N	**recall**	**precision**	**recall rate**
1	0.119	0.980	0.980
3	0.284	0.800	1.000
5	0.395	0.678	1.000
7	0.456	0.577	1.000
10	0.495	0.436	1.000
13	0.509	0.345	1.000
15	0.519	0.305	1.000
17	0.534	0.281	1.000
20	0.558	0.250	1.000

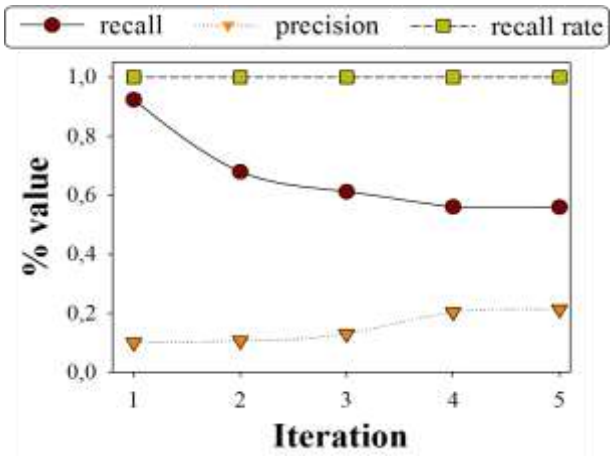

Figure 53 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of Graphics category.

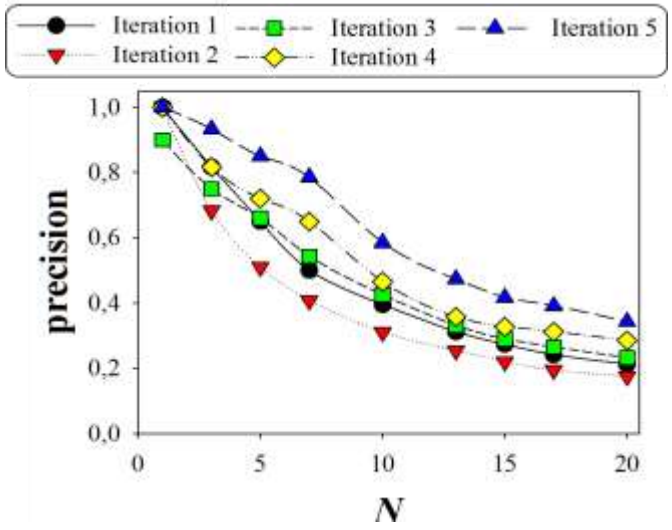

Figure 54 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Graphics category.

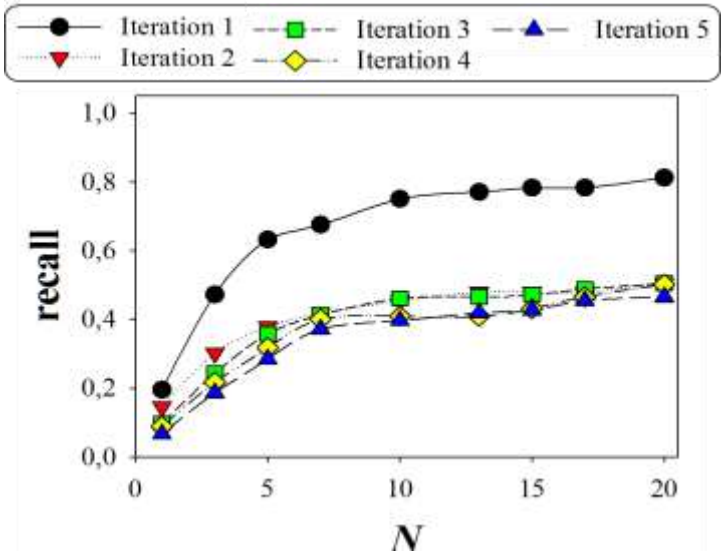

Figure 55 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Graphics category.

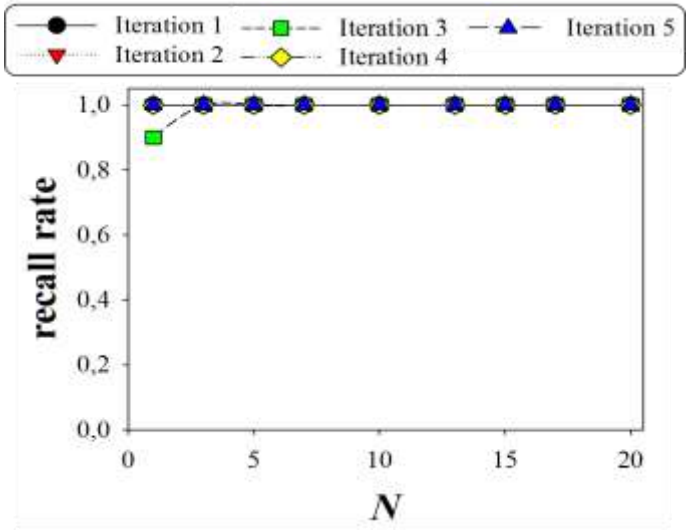

Figure 56 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Graphics category.

Table 48 APIs recommended in top-20 lists in Stage B of Graphics category

#	API	Frequency	#	API	Frequency
1	javax.swing.table	1	19	java.awt.datatransfer	13
2	org.lwjgl.util.vector	1	20	java.awt.image	14
3	org.lwjgl.opengl.glu	1	21	java.util.zip	14
4	java.nio.channels	2	22	java.awt.font	14
5	org.apache.log4j	2	23	javax.swing.event	14
6	javax.swing.plaf	3	24	javax.imageio.stream	15
7	java.awt.print	4	25	javax.swing.text	15
8	java.lang.reflect	6	26	java.text	15
9	javax.swing	6	27	java.nio	16
10	java.applet	7	28	java.awt.geom	16
11	java.math	7	29	javax.imageio	16
12	java.util	8	30	java.awt	16
13	org.w3c.dom	10	31	javax.swing.tree	16
14	java.io	10	32	org.xml.sax	17
15	javax.swing.text.html	12	33	java.awt.event	17
16	java.beans	13	34	javax.swing.border	17
17	java.util.regex	13	35	java.net	18
18	javax.vecmath	13	36	javax.swing.filechooser	18

After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. As in Stage B of API recommendation we made

five replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value. In this category, we found that replicates 5, 3, 5, 2 and 3 presented best recall values for iterations 1, 2, 3, 4, and 5 respectively. In Table 48, we showed the number of APIs recommended, APIs, and frequency value, where frequency is the number of times the API was recommended in a top-20 over the 20 target software.

6.6 Home & Education category

We found 20 relevant Java software in Sourcerer dataset corresponding exclusively to Home & Education SourceForge category. In Table 49, we showed those software and their number of APIs.

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Home & Education category of the stage of software development, we did the 5-fold cross validation (Figure 57). Hence, in every iteration we used four different software (i.e., 20% of data) as test set and 16 software (i.e., 80% of data) as training set. In that partitioning, we avoided overlapping, i.e., every software appears just once in test set. In this partitioning process, we saved `.wst` files of every iteration.

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation methodology in every stage of API recommendations. We exposed these results as follows.

6.6.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the

5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 50). The *minSupport* = 0.1 presented the highest recall. Hence, we chose it as the "best" *minSupport value* for FIS technique in Home & Education category.

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the Home & Education category in Stage A of development when receiving large lists of APIs recommended as well as when receiving top-*N* list of APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Table 49 Baseline data for Home & Education category

#	SOFTWARE NAME	NUMBER OF APIs
1	Populationsentwicklung	5
2	mbells	7
3	DukeBot-Beta	8
4	Jding_2	9
5	MyComp	10
6	democracy-core	10
7	org.hanyudictionary	11
8	struktor.svn.sf.net	12
9	RaceTrack	12
10	SeePeople	13
11	kiga3000	13
12	jipsi	14
13	backend	17
14	e-sim	19
15	JReportingGrid	21
16	coursework2_5	33
17	cplab	38
18	avatal_sf_sept04	41
19	CabaWeb	43
20	NewEledge	75

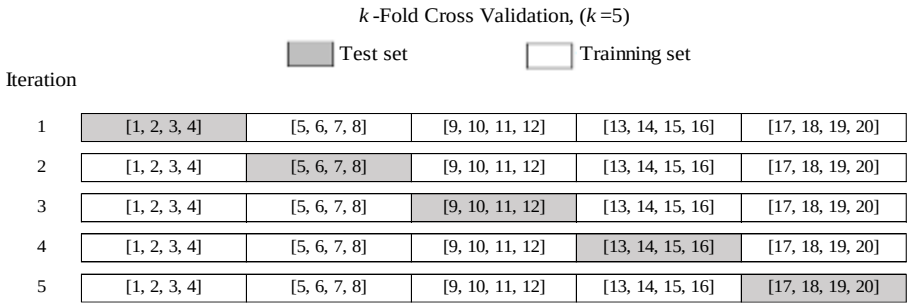

Figure 57 Home & Education category dataset partition

Table 50 Effect of varying *minSupport* value in top-10 lists of APIs recommended in Stage A of Home & Education category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.570	0.224	1.000
0.2	0.500	0.413	1.000
0.3	0.465	0.501	1.000
0.4	0.371	0.634	1.000
0.5	0.355	0.707	1.000
0.6	0.305	0.735	1.000
0.7	0.158	0.838	1.000
0.8	0.142	0.950	1.000
0.9	0.112	0.950	1.000

Regarding the evaluation when receiving lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 51). In addition, for analyzing every iteration, we exposed evaluation results in Figure 58.

Table 51 Results of evaluation metrics for large lists of APIs recommended in Stage A of Home & Education category.

recall	**precision**	**recall rate**
0.570	0.224	1.000

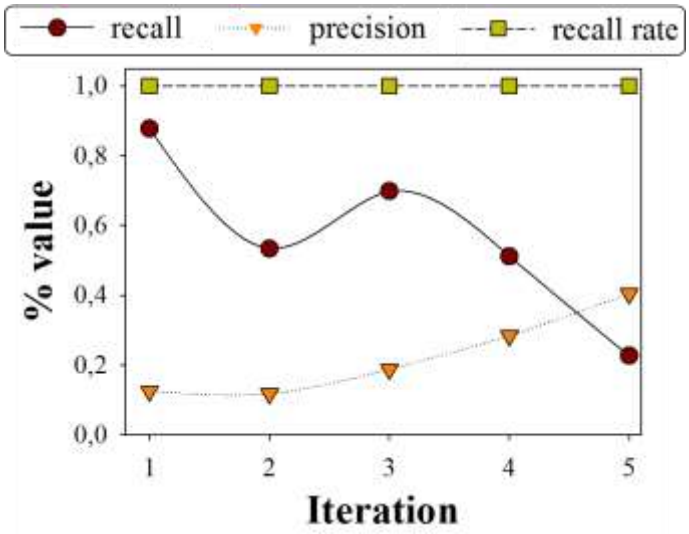

Figure 58 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of Home & Education category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 52). In addition, for analyzing every iteration, we exposed the results of precision (Figure 59), recall (Figure 60), and recall rate (Figure 61) for each iteration.

Table 52 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of Home & Education category.

N	**recall**	**precision**	**recall rate**
1	0.077	1.000	1.000
3	0.184	0.817	1.000
5	0.284	0.740	1.000
7	0.360	0.679	1.000
10	0.434	0.585	1.000
13	0.463	0.496	1.000
15	0.491	0.463	1.000
17	0.517	0.435	1.000
20	0.533	0.390	1.000

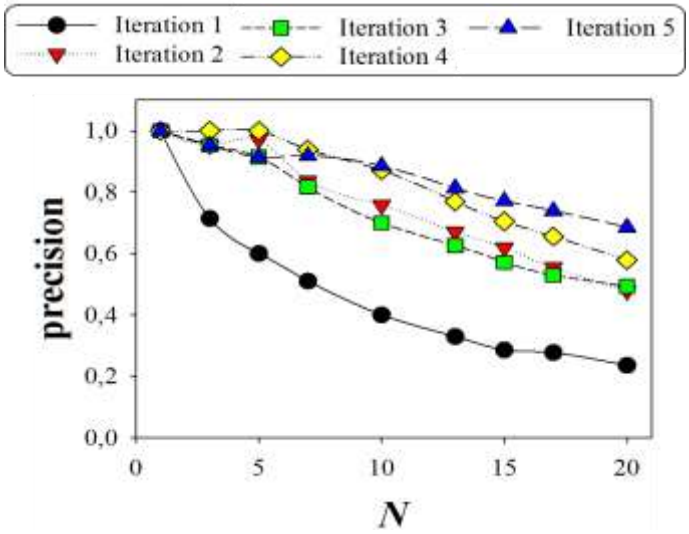

Figure 59 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Home & Education category.

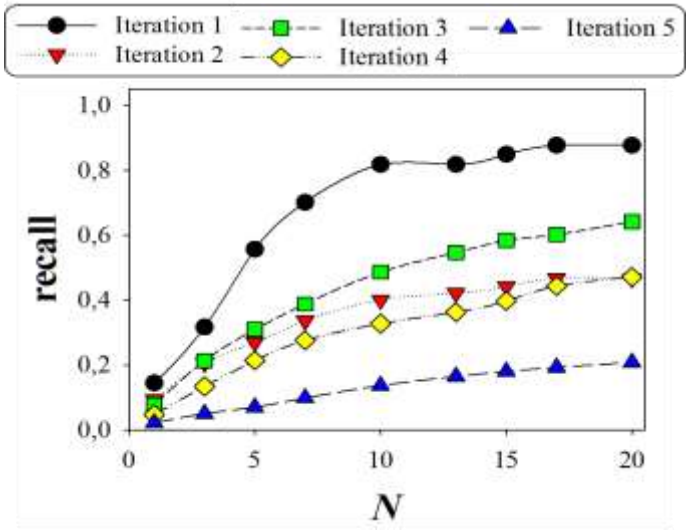

Figure 60 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Home & Education category.

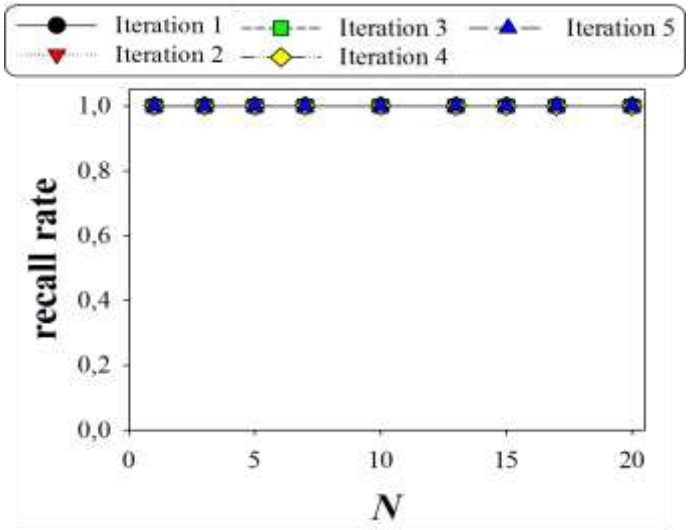

Figure 61 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Home & Education category.

After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. In Table 53, we showed number of recommended APIs, their name and their frequency value, where frequency is the number of times the API was recommended over the five iterations.

Table 53 APIs recommended in top-20 lists in Stage A of Home & Education category

#	API	Frequency
1	java.beans	1
2	java.util.regex	1
3	javax.swing.filechooser	1
4	java.awt.print	2
5	java.util.logging	3
6	javax.servlet.http	3
7	java.awt.geom	3
8	javax.mail.internet	4
9	javax.swing.text	4
10	javax.servlet	4
11	junit.framework	4
12	java.sql	5
13	java.util	5
14	java.awt.image	5
15	javax.swing.table	5
16	java.util.zip	5
17	java.net	5
18	java.lang.reflect	5
19	javax.swing.event	5
20	java.awt	5
21	java.text	5
22	java.io	5
23	java.awt.event	5
24	javax.swing.border	5
25	javax.swing	5

6.6.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found for FIS technique in Stage A, since we used the same data sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 54) and we found that varying *k* in 5 presented the best recall and precision values. Consequently, we chose *minSupport* = 0.1 for FIS technique and *k* = 5 for CF technique in Home & Education category,

Table 54 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of Home & Education category.

k	**recall**	**precision**	**recall rate**
5	0.491	0.363	0.950
10	0.490	0.359	0.950
15	0.481	0.350	0.950

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in Home & Education category and Stage B of development when receiving lists of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In case of top-*N*, we evaluated the effect of varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations and replicates of the 5-fold cross validation (Table 55). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 62.

Table 55 Results of evaluation metrics for large lists of APIs recommended in Stage B of Home & Education category.

recall	precision	recall rate
0.500	0.268	1.000

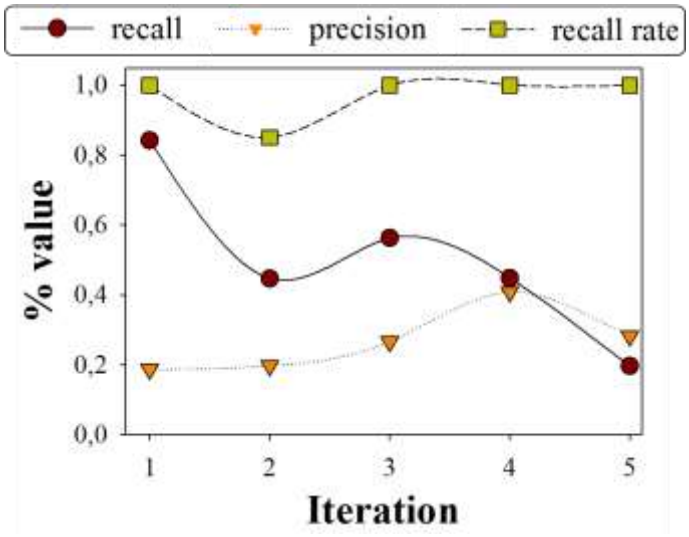

Figure 62 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of Home & Education category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 56). In addition, for analyzing every iteration, we expose average results of precision (Figure 63), recall (Figure 64), and recall rate (Figure 65) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly (Section 5.4).

Table 56 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of Home & Education category.

N	**recall**	**precision**	**recall rate**
1	0.127	0.870	0.870
3	0.303	0.693	0.920
5	0.368	0.516	0.960
7	0.412	0.420	0.970
10	0.464	0.342	0.970
13	0.490	0.307	0.970
15	0.493	0.290	0.970
17	0.496	0.281	0.970
20	0.498	0.275	0.970

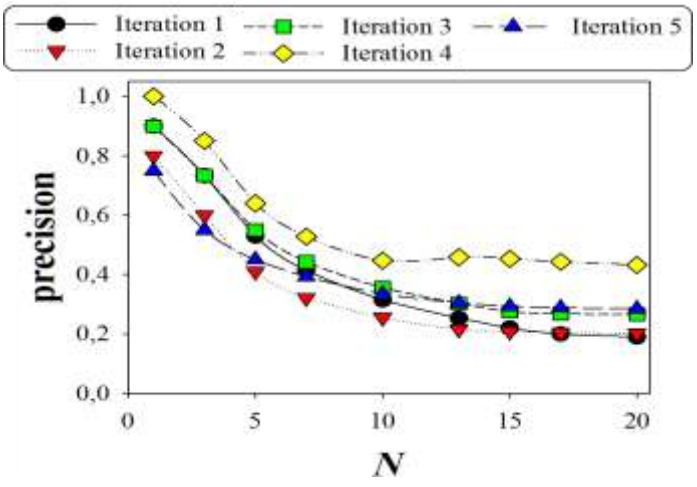

Figure 63 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Home & Education category.

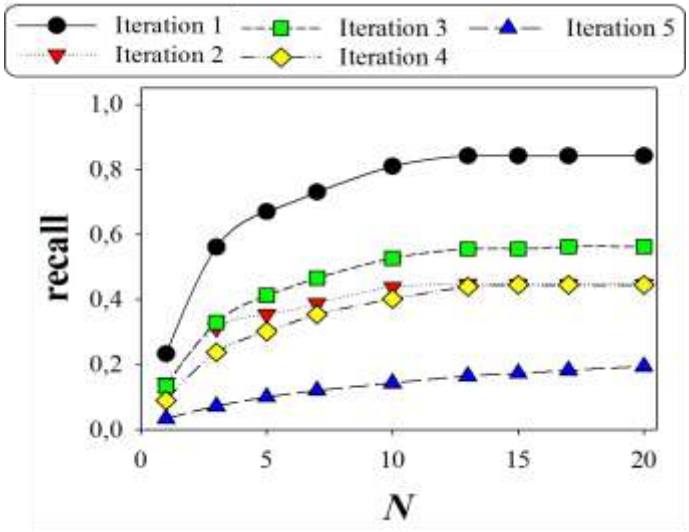

Figure 64 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Home & Education category.

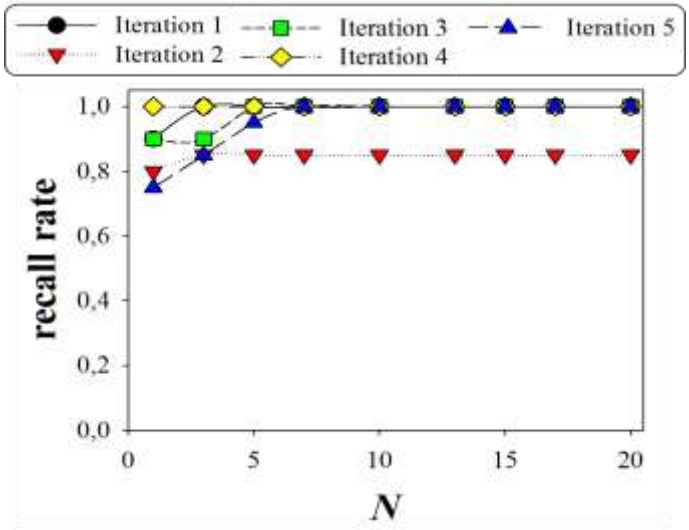

Figure 65 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Home & Education category.

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. As in Stage B of API recommendation we made five

replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value. In this category, we found that replicates 1, 4, 3, 2, and 1 presented the best recall values for iterations 1, 2, 3, 4, and 5 respectively. In Table 57, we showed the number of APIs recommended, the APIs, and frequency value, where frequency is the number of times the API was recommended in a top-20 list over the 20 target software.

Table 57 APIs recommended in top-20 lists in Stage B of Home & Education category

#	API	Frequency
1	org.apache.struts.action	1
2	org.jdom	1
3	javax.sql	1
4	javax.servlet.jsp	1
5	javax.mail	1
6	org.apache.commons.logging	1
7	javax.activation	1
8	javax.servlet.jsp.tagext	1
9	org.apache.struts.util	1
10	javax.xml.transform.stream	1
11	org.apache.struts.validator	1
12	org.apache.struts	1
13	junit.framework	2
14	javax.servlet.http	3
15	java.util.Map	3
16	java.math	4
17	javax.servlet	4
18	java.util.zip	5
19	java.util.regex	5
20	javax.print.attribute.standard	6
21	org.apache.log4j	6
22	javax.print	7
23	javax.print.attribute	7
24	javax.swing.filechooser	9
25	java.awt.geom	9
26	javax.swing	9
27	javax.swing.plaf	10
28	java.util.logging	11
29	java.beans	11
30	java.io	11
31	java.awt.print	12
32	java.net	13
33	java.awt	13
34	java.sql	14
35	java.util	14
36	java.text	14
37	java.awt.event	15
38	javax.swing.border	15
39	javax.swing.table	16
40	java.lang.reflect	16
41	javax.swing.text	16
42	java.awt.image	17
43	javax.swing.event	19

6.7 Science & Engineering category

We found 50 relevant Java software in Sourcerer dataset corresponding exclusively to Science & Engineering SourceForge category. In Table 58, we showed those software and their number of APIs.

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Science & Engineering category

regardless of the stage of software development, we did the 5-fold cross validation (Figure 66). Hence, in every iteration we used 10 different software (i.e., 20% of data) as test set and 40 software (i.e., 80% of data) as training set. In that partitioning, we avoided overlapping, i.e., every software appears just once in the test set. Besides, we saved `.wst` files of every iteration.

Table 58 Baseline data for Science & Engineering category

#	SOFTWARE NAME	NUMBER OF APIs	#	SOFTWARE NAME	NUMBER OF APIs
1	jPicProgrammer	5	26	Impact	22
2	neuralj	6	27	RemoteMaster	22
3	Diabetes	7	28	JSpecView-BH	26
4	jVisualizer	7	29	seqtracs	27
5	WebTranslator	7	30	j2eeweather	28
6	GPSBlender	8	31	jplot	28
7	portlets	8	32	repast-jelly-taglibrary	28
8	jdwglib	9	33	SDD1	28
9	projectlima	10	34	vwtk	29
10	MolTools Core	11	35	GT	30
11	nudj	11	36	medSLT	30
12	C4Jadex	12	37	org.jactr.eclipse.production	33
13	esra	12	38	sntool	33
14	jef	12	39	XQTav	34
15	cifdom	13	40	jasa	39
16	Flicker	14	41	ProbeMaker	39
17	_mbfuzzit	15	42	yawn	40
18	JavaHMI	15	43	vp	40
19	ili2sql	16	44	CircuitSmith	43
20	jwnl	16	45	mdr-sourceforge	45
21	SaukhyaLite	16	46	j-Algo	51
22	joonegap	17	47	CyClone_Core	52
23	brunswick	21	48	ASDN	63
24	appl.poz5.rw	21	49	mymaps	71
25	ecolosim-cvs2	22	50	UnBBayesOntology	73

In order to evaluate our recommendation methodology for Software Engineers with software categorized in Science & Engineering category regardless of the stage of software development, we did the 5-fold cross validation (Figure 66). Hence, in every iteration we used 10 different software (i.e., 20% of data) as test set and 40 software (i.e., 80% of data) as training set. In that

partitioning, we avoided overlapping, i.e., every software appears just once in the test set. Besides, we saved `.wst` files of every iteration.

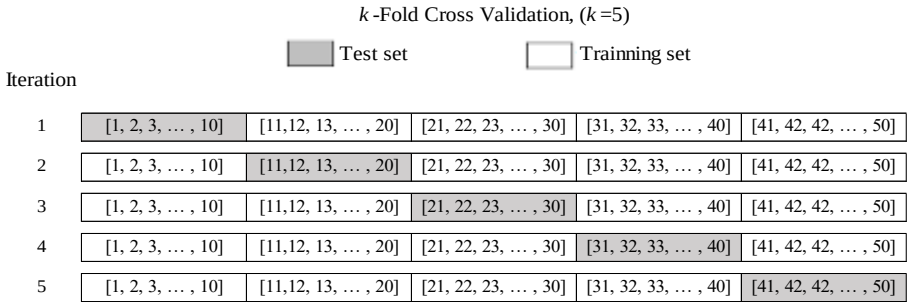

Figure 66 Science & Engineering category dataset partition

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation methodology in every stage of API recommendations. We exposed these results as follows.

6.7.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 59). From 0.1 to 0.3, *minSupport* presented same results with the highest recall. Hence, we

chose *minSupport* = 0.1 as the "best" *minSupport value* for FIS technique in Science & Engineering category, since choosing other *minSupport* value did not mean any improvement either for precision or recall rate values.

Table 59 Effect of varying *minSupport* value in top-10 lists of APIs recommended in Stage A of Science & Engineering category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.324	0.628	1.000
0.2	0.324	0.628	1.000
0.3	0.324	0.628	1.000
0.4	0.319	0.638	1.000
0.5	0.280	0.711	1.000
0.6	0.233	0.786	1.000
0.7	0.187	0.836	1.000
0.8	0.137	0.903	1.000
0.9	0.117	0.970	1.000

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the Science & Engineering category and in Stage A of development when receiving large lists of APIs recommended as well as when receiving top-*N* list of APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding evaluation when receiving large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Table 60). In addition, for analyzing every iteration, we exposed evaluation results (Figure 67).

Table 60 Results of evaluation metrics for large lists of APIs recommended in Stage A of Science & Engineering category.

recall	**precision**	**recall rate**
0.555	0.286	1.000

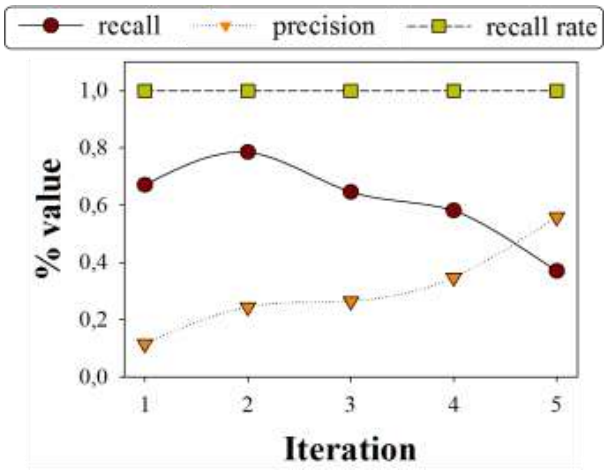

Figure 67 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of Science & Engineering category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 61). In addition, for analyzing every iteration, we exposed the results of precision (Figure 68), recall (Figure 69), and recall rate (Figure 70) for each iteration.

Table 61 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of Science & Engineering category.

N	**recall**	**precision**	**recall rate**
1	0.058	0.980	0.980
3	0.154	0.880	1.000
5	0.225	0.812	1.000
7	0.279	0.740	1.000
10	0.324	0.628	1.000
13	0.376	0.583	1.000
15	0.403	0.552	1.000
17	0.418	0.511	1.000
20	0.437	0.458	1.000

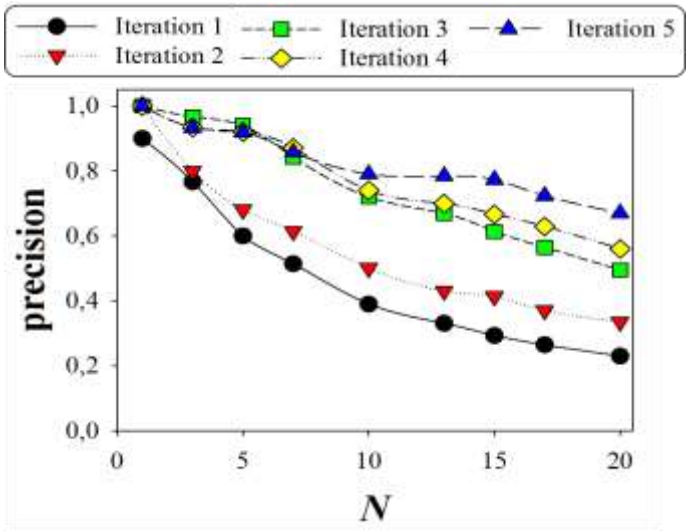

Figure 68 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Science & Engineering category.

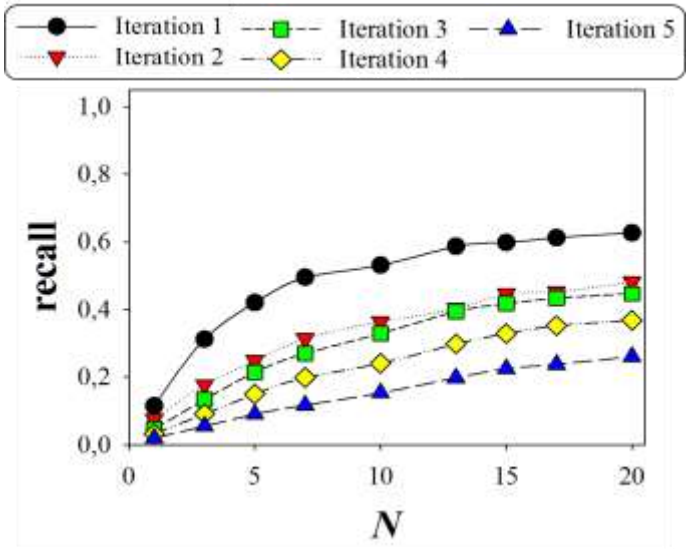

Figure 69 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Science & Engineering category.

After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. In Table 62, we showed number of recommended

APIs, their name and their frequency value, where frequency is the number of times the API was recommended over the five iterations.

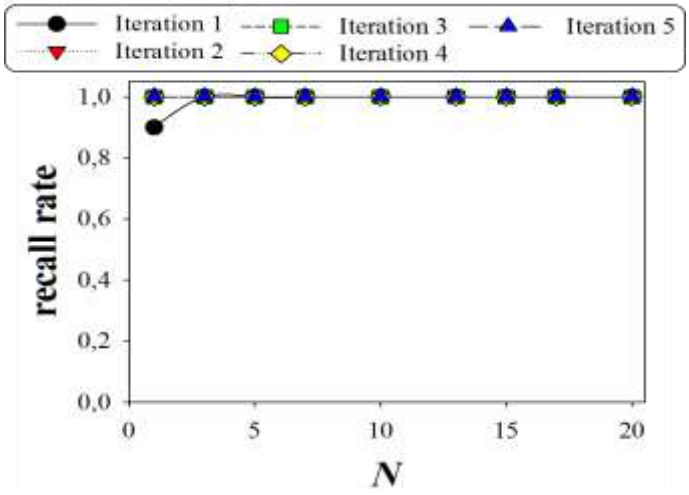

Figure 70 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of Science & Engineering category.

Table 62 APIs recommended in top-20 lists in Stage A of Science & Engineering category

#	API	Frequency
1	java.util.zip	1
2	org.apache.commons.logging	1
3	java.lang	1
4	org.apache.log4j	1
5	javax.xml.parsers	2
6	javax.swing.tree	2
7	java.util.regex	3
8	org.xml.sax	4
9	java.util	5
10	java.awt.image	5
11	java.beans	5
12	javax.swing.table	5
13	java.net	5
14	javax.swing.filechooser	5
15	java.lang.reflect	5
16	java.awt.geom	5
17	javax.swing.event	5
18	javax.swing.text	5
19	java.awt	5
20	java.text	5
21	java.io	5
22	java.awt.event	5
23	javax.swing.border	5
24	junit.framework	5
25	javax.swing	5

6.7.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-

fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found for FIS technique in Stage A, since we used the same data sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 63) and we found that varying *k* in 25 presented the best recall and precision values. Consequently, we chose *minSupport* = 0.1 for FIS technique and *k* = 25 for CF technique in Science & Engineering category,

Table 63 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of Science & Engineering category.

k	**recall**	**precision**	**recall rate**
5	0.380	0.407	0.976
10	0.389	0.415	0.976
15	0.389	0.417	0.968
20	0.390	0.418	0.972
25	0.392	0.419	0.972

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in Science & Engineering category and Stage B of development when receiving

large lists of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In the case of top-*N*, we evaluated the effect of varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when receiving large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations and replicates of the 5-fold cross validation (Table 64). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 71.

Table 64 Results of evaluation metrics for large lists of APIs recommended in Stage B of Science & Engineering category.

recall	precision	recall rate
0.529	0.198	0.984

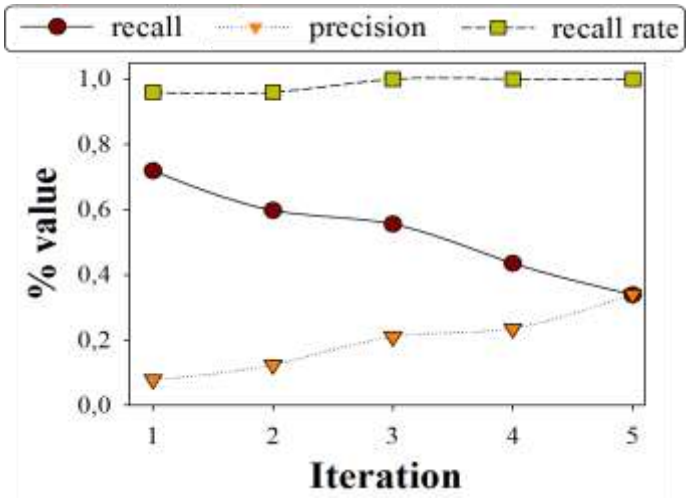

Figure 71 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of Science & Engineering category.

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 65). In addition, for analyzing

every iteration, we expose average results of precision (Figure 72), recall (Figure 73), and recall rate (Figure 74) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly (Section 5.4).

Table 65 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of Science & Engineering category.

N	**recall**	**precision**	**recall rate**
1	0.101	0.908	0.908
3	0.224	0.712	0.944
5	0.287	0.582	0.964
7	0.333	0.499	0.976
10	0.382	0.410	0.980
13	0.408	0.341	0.980
15	0.420	0.306	0.980
17	0.436	0.282	0.980
20	0.460	0.255	0.980

After obtaining the evaluation results, we identified APIs recommended in top-20 over all iterations. As in Stage B of API recommendation we made five replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value. In this category, we found that replicates 2, 5, 1, 1, and 3 presented the best recall values for iterations 1, 2, 3, 4, and 5 respectively. In Table 66, we showed the number of APIs recommended, the APIs, and frequency value, where frequency is the number of times the API was recommended in a top-20 list over the 50 target software.

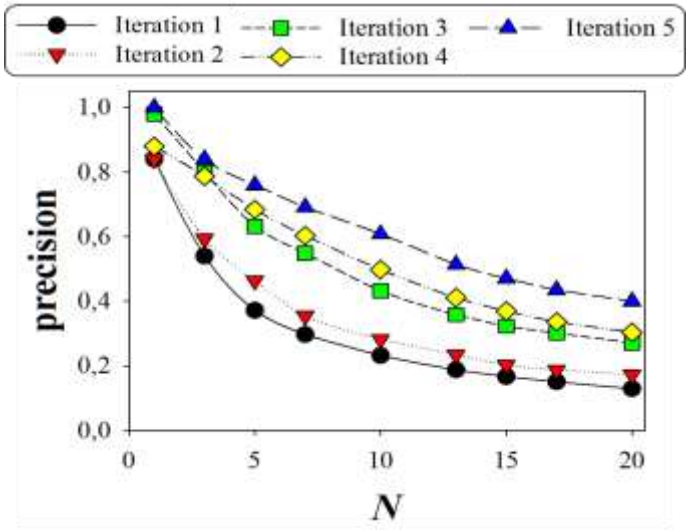

Figure 72 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Science & Engineering category.

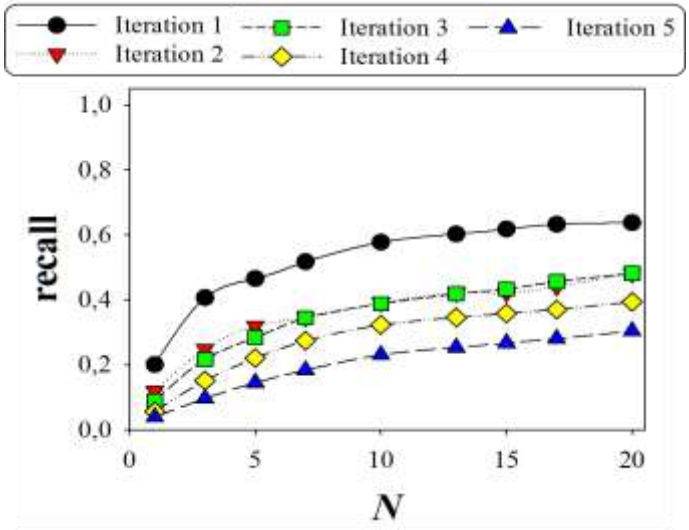

Figure 73 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Science & Engineering category.

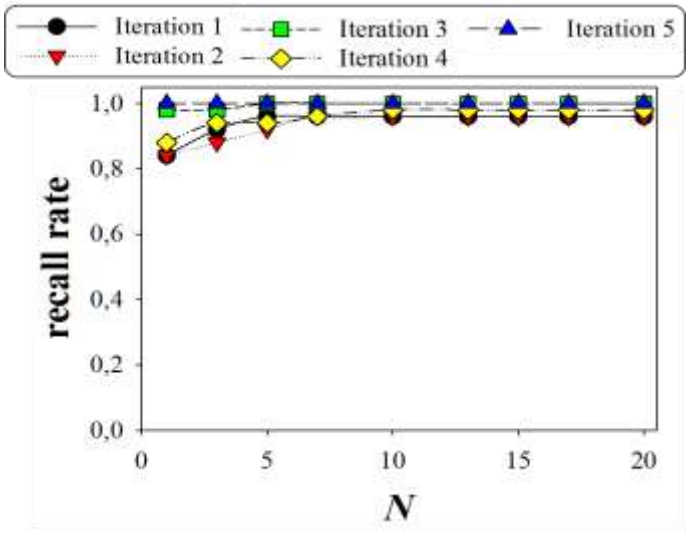

Figure 74 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of Science & Engineering category.

Table 66 APIs recommended in top-20 lists in Stage B of Science & Engineering category

#	API	Frequency
1	java.util.logging	1
2	java.applet	1
3	javax.xml.transform	1
4	org.apache.commons.logging	2
5	java.awt.datatransfer	4
6	javax.swing.plaf	7
7	java.nio	9
8	java.awt.font	12
9	org.w3c.dom	13
10	java.sql	14
11	javax.swing	15
12	javax.imageio	18
13	java.awt.print	21
14	java.lang	22
15	java.math	23
16	java.util.zip	27
17	java.io	27
18	org.apache.log4j	33
19	org.xml.sax	34
20	java.awt	34
21	javax.xml.parsers	35
22	java.awt.event	36
23	java.net	37
24	javax.swing.event	37
25	java.text	38
26	java.util	39
27	java.awt.geom	39
28	javax.swing.filechooser	40
29	java.lang.reflect	40
30	java.util.regex	41
31	javax.swing.border	41
32	java.awt.image	43
33	java.beans	43
34	javax.swing.text	43
35	javax.swing.tree	43
36	junit.framework	43
37	javax.swing.table	44

6.8 System Administration category

We found 35 relevant Java software in Sourcerer dataset corresponding exclusively to System Administration SourceForge category. In Table 67, we showed those software and their number of APIs.

In order to evaluate our recommendation methodology for Software Engineers with software categorized in System Administration category regardless of the stage of software development, we did the 5-fold cross validation (Figure 75). Hence, in every iteration we used seven different software (i.e., 20% of data) as test set and 28 software (i.e., 80% of data) as training set. In that partitioning, we avoided overlapping, i.e., every software appears just once in test set. In this partitioning process, we saved `.wst` files of every iteration.

Table 67 Baseline data for System Administration category

#	SOFTWARE NAME	NUMBER OF APIs	#	SOFTWARE NAME	NUMBER OF APIs
1	filesystemscanner	7	19	logshark	16
2	joesnmp	5	20	gltmon	17
3	Netinfo	5	21	logtail	17
4	RegExSearchReplace	6	22	JNetMap	18
5	NativeCall	7	23	MadCommander	19
6	remotetea	7	24	Route64	19
7	JHexDump	7	25	ExtendendRootTree	22
8	hotpotato	8	26	LdapUserEditor	22
9	oxygen	8	27	JBManageIT	22
10	UDPReader	8	28	eclipservices	25
11	in1660	9	29	pct4g	25
12	JZipUpdate	10	30	SysFrame	31
13	tresMonitor	11	31	Tests	34
14	xgridagent	12	32	scytha	42
15	MOSInstaller	13	33	jmanage	58
16	soap-stone	13	34	HermesJMS	117
17	syrup	14	35	WSMX	147
18	Kurumix	15			

The strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our

recommendation methodology in every stage of API recommendations. We exposed these results as follows.

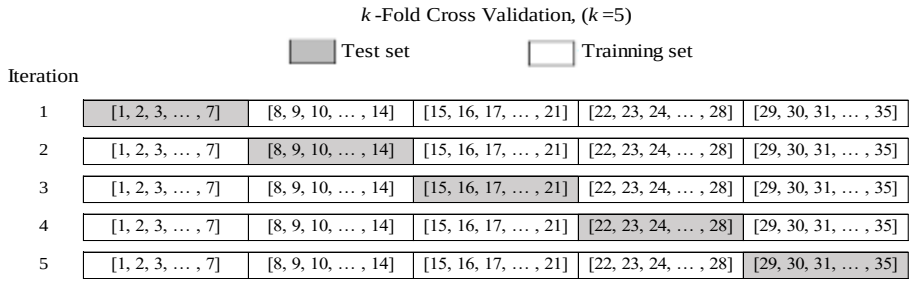

Figure 75 System Administration category dataset partition

6.8.1 API recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. In this Stage, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Thereby, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 68). The *minSupport* in 0.1 and 0.2 presented same results with the highest recall. Hence, we chose *minSupport* = 0.1 as the "best" *minSupport value* for FIS technique in System Administration category, since choosing *minSupport* = 0.2 did not mean any significant improvement either for precision or recall rate values.

After finding *minSupport* value, we used it for evaluating our methodology for recommending APIs to Software Engineers whose software was in the System Administration category and in Stage A of development when receiving large lists of APIs recommended as well as when receiving top-*N* list of

APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Table 68 Effect of varying *minSupport* value in top-10 lists of APIs recommended in Stage A of System Administration category.

minSupport	**recall**	**precision**	**recall rate**
0.1	0.397	0.543	1.000
0.2	0.397	0.543	1.000
0.3	0.393	0.553	1.000
0.4	0.328	0.661	1.000
0.5	0.282	0.704	1.000
0.6	0.212	0.886	1.000
0.7	0.200	0.900	1.000
0.8	0.158	0.971	1.000
0.9	0.158	0.971	1.000

Regarding the evaluation when receiving lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Table 69). In addition, for analyzing every iteration, we exposed evaluation results in Figure 76.

Table 69 Results of evaluation metrics for large lists of APIs recommended in Stage A of System Administration category.

recall	**precision**	**recall rate**
0.628	0.231	1.000

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 70). In addition, for analyzing every iteration, we exposed the results of precision (Figure 77), recall (Figure 78), and recall rate (Figure 79) for each iteration.

After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. In Table 71, we showed number of recommended

APIs, their name and their frequency value, where frequency is the number of times the API was recommended over the five iterations.

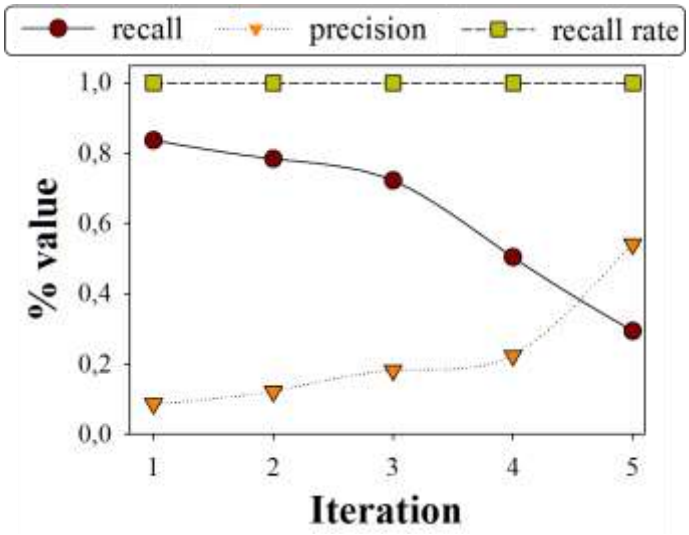

Figure 76 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage A of System Administration category.

Table 70 Results of evaluation metrics when varying *N* of top-*N* lists in Stage A of System Administration category.

N	recall	precision	recall rate
1	0.077	0.943	0.971
3	0.212	0.886	1.000
5	0.273	0.714	1.000
7	0.332	0.637	1.000
10	0.397	0.543	1.000
13	0.420	0.455	1.000
15	0.437	0.417	1.000
17	0.462	0.397	1.000
20	0.521	0.379	1.000

6.8.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-

fold cross validation method. Therefore, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

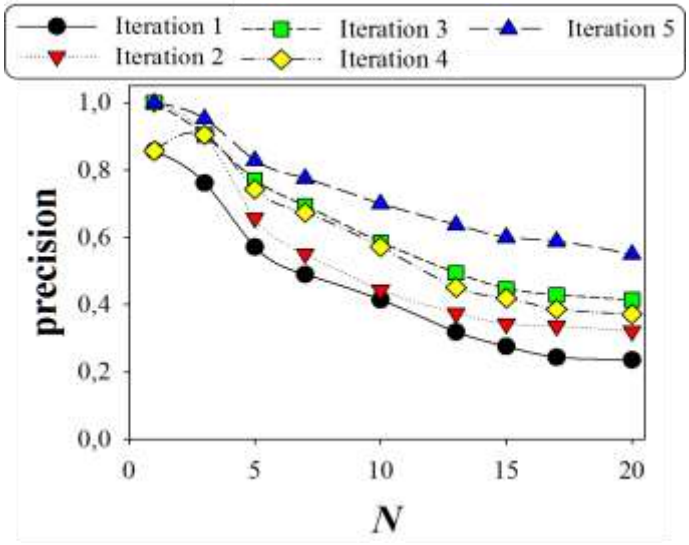

Figure 77 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of System Administration category.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found for FIS technique in Stage A, since we used the same data sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 72) and we found that varying *k* in 5 presented the

best recall and precision values. Consequently, we chose *minSupport* = 0.1 for FIS technique and *k* = 5 for CF technique in System Administration category.

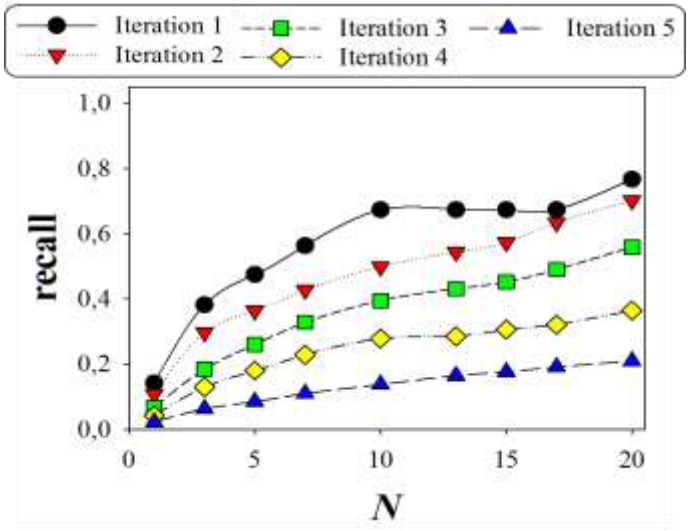

Figure 78 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of System Administration category.

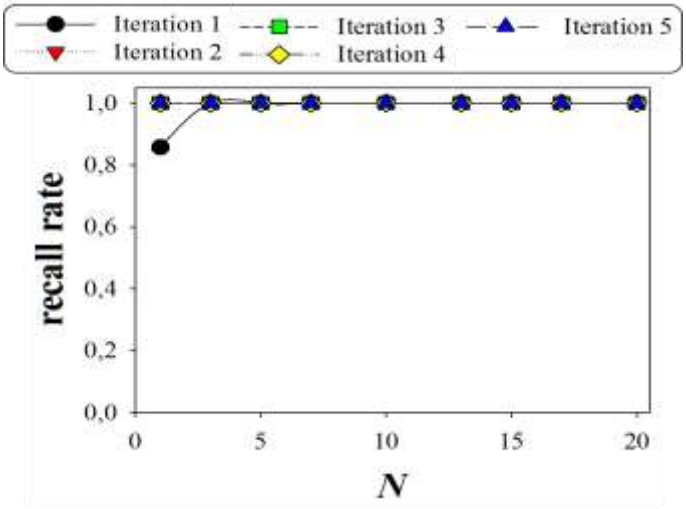

Figure 79 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage A of System Administration category.

Table 71 APIs recommended in top-20 lists in Stage A of System Administration category

#	API	Frequency	#	API	Frequency
1	java.sql	1	13	java.util	5
2	java.awt.datatransfer	1	14	javax.swing.event	5
3	javax.swing.filechooser	1	15	java.util.zip	5
4	java.nio	3	16	java.awt	5
5	java.util.logging	4	17	java.text	5
6	org.w3c.dom	4	18	java.net	5
7	javax.swing.table	4	19	java.util.regex	5
8	javax.xml.parsers	4	20	java.io	5
9	org.xml.sax	4	21	java.awt.event	5
10	javax.swing.border	4	22	java.util.jar	5
11	java.lang.reflect	5	23	junit.framework	5
12	java.security	5	24	javax.swing	5

Table 72 Effect of varying *k*, i.e., the number of nearest neighbors in top-10 lists of APIs recommended in Stage B of System Administration category.

k	**recall**	**precision**	**recall rate**
5	0.459	0.343	0.983
10	0.435	0.322	0.971
15	0.438	0.325	0.971
20	0.431	0.318	0.983
25	0.432	0.318	0.977

After setting *minSupport* and *k* values, we used them for evaluating our methodology for recommending APIs to Software Engineers whose software was in System Administration category and Stage B of development when receiving large lists of APIs recommended as well as when receiving top-*N* lists of APIs recommended. In the case of top-*N*, we evaluated the effect by varying *N* in 1, 3, 5, 7, 10, 13, 15, 17, and 20.

Regarding the evaluation when large lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations and replicates of the 5-fold cross validation (Table 73). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 80.

Table 73 Results of evaluation metrics for large lists of APIs recommended in Stage B of System Administration category.

recall	precision	recall rate
0.529	0.274	0.994

On the other hand, when receiving top-*N* lists of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from all of iterations of the 5-fold cross validation (Table 74). In addition, for analyzing every iteration, we expose average results of precision (Figure 81), recall (Figure 82), and recall rate (Figure 83) for each iteration. In this stage, it is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly (Section 5.4).

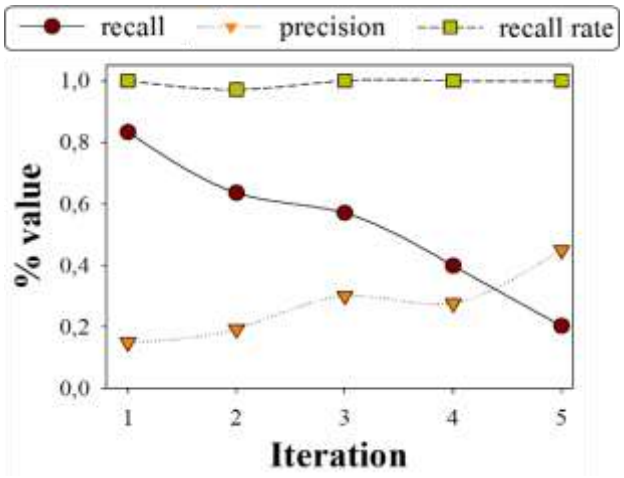

Figure 80 Results of evaluation metrics for every iteration of 5-fold cross validation for large lists of APIs recommended in Stage B of System Administration category.

After obtaining evaluation results, we identified APIs recommended in top-20 over all iterations. As in Stage B of API recommendation we made five replicates of the recommendation test for every target software, we decided to select top-20 lists from the replicate with best recall value. In this category, we found that replicates 1, 3, 3, 1 and 2 presented the best recall values for iterations

1, 2, 3, 4, and 5 respectively. In Table 75, we showed the number of APIs recommended, the APIs, and frequency value, where frequency is the number of times the API was recommended in a top-20 over the 35 target software.

Table 74 Results of evaluation metrics when varying *N* of top-*N* lists in Stage B of System Administration category.

N	recall	precision	recall rate
1	0.134	0.891	0.891
3	0.292	0.672	0.971
5	0.373	0.520	0.983
7	0.414	0.425	0.983
10	0.456	0.350	0.989
13	0.493	0.308	0.994
15	0.512	0.295	0.994
17	0.523	0.287	0.994
20	0.527	0.279	0.994

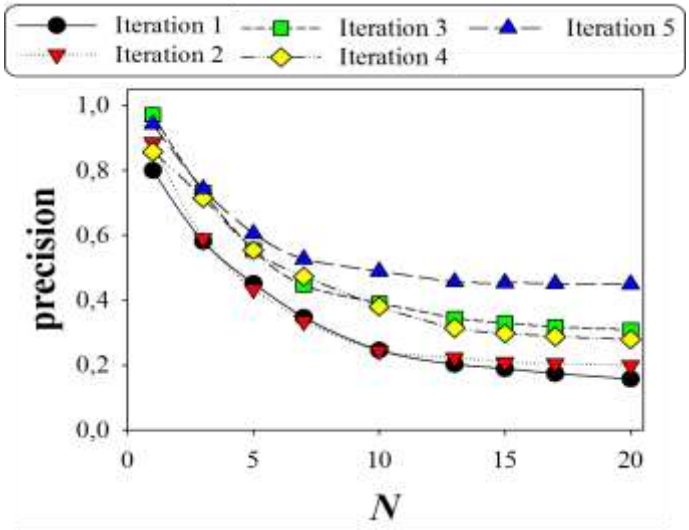

Figure 81 Precision metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of System Administration category.

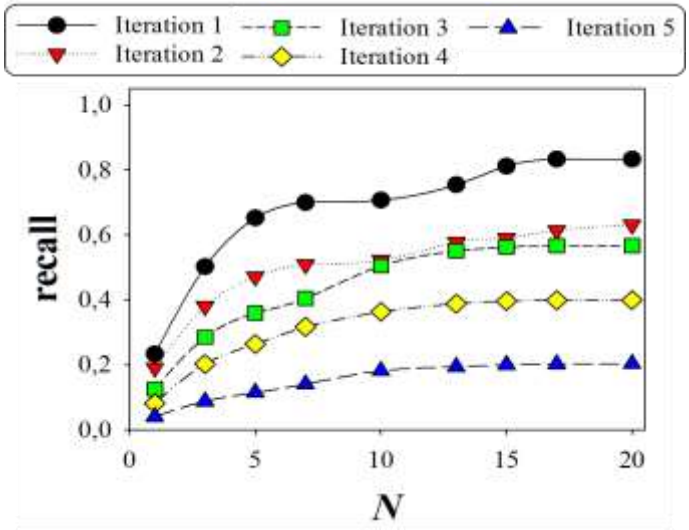

Figure 82 Recall metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of System Administration category.

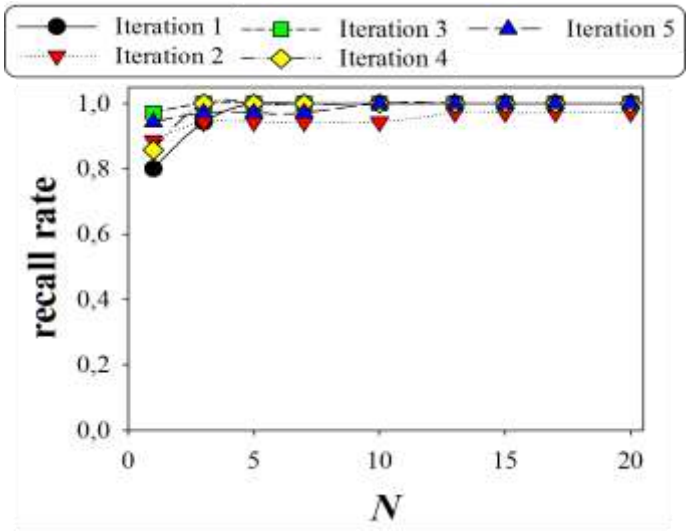

Figure 83 Recall rate metric for every iteration of 5-fold cross validation when varying *N* of top-*N* lists in Stage B of System Administration category.

6.9 General results

We found 315 relevant Java software in Sourcerer dataset corresponding exclusively to every SourceForge category. As we exposed from Section 7.1 to Section 7.8, regardless of the stage of software development, we evaluated our recommendation methodology for every category using the 5-fold cross validation (i.e., in every iteration we used 20% of software as test set and the remaining 80% of software as training set).

In addition, the strategy for simulating Software Engineers' behavior changes depending on the stage of software development (Section 5.4). Hence, we applied the evaluation strategy for Stage A and Stage B and we computed and averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation methodology in every stage of API recommendations.

Table 75 APIs recommended in top-20 lists in Stage B of System Administration category

#	API	Frequency	#	API	Frequency
1	javax.management	1	20	javax.swing.tree	17
2	java.rmi.registry	1	21	java.util.logging	18
3	java.rmi.server	2	22	java.security	18
4	java.rmi	3	23	javax.swing.filechooser	19
5	javax.xml.transform.dom	4	24	junit.framework	19
6	java.math	4	25	javax.swing.event	21
7	javax.xml.transform	4	26	java.io	21
8	javax.xml.transform.stream	4	27	java.awt.event	21
9	org.xml.sax	6	28	javax.swing.table	23
10	java.beans	7	29	javax.swing.border	23
11	javax.swing.text	7	30	java.util	24
12	javax.naming	9	31	java.lang.reflect	24
13	java.nio.channels	10	32	java.nio	25
14	java.lang	10	33	java.net	25
15	java.sql	12	34	java.util.regex	25
16	javax.xml.parsers	12	35	java.util.jar	25
17	java.awt.datatransfer	12	36	java.util.zip	26
18	org.w3c.dom	13	37	java.awt	26
19	javax.swing	16	38	java.text	32

Then, in this section, we exposed averaged results considering the eight categories. These results are presented as follows.

6.9.1 API Recommendation for Stage A

Stage A consists on Software Engineers in initial stage of software development, i.e., software do not use APIs. For every category of Stage A, in every iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software.

In Stage A, the core is the FIS technique (Section 5.3). Thus, we needed to find the "best" threshold value, i.e., the "best" *minSupport value*. Then, we used the plug-in and analyzed the effect of varying the *minSupport* value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended and we found that from 7 of the 8 categories, the best *minSupport* value was of 0.1, excepting Business & Enterprise category where we found best *minSupport* value of 0.4.

Thus, as averaged results regarding large lists in Stage A for the eight categories, we found a recall of 56.7%, a precision of 31.3%, and a recall rate of 99.8%. On the other hand, regarding top-20 lists in Stage A for the eight categories, we found a recall 47.1%, a precision of 45.6%, and a recall rate of 99.8%. We discussed these results in Section 8.9.

Furthermore, to identify the most and less APIs recommended in Stage A among the eight categories, we established a 4-point rating scale (R, P, L, and F):

a) Rarely recommended (R) represents that an API was recommended in 1 or 2 categories;
b) Partially recommended (P) represents that an API was recommended in 3 or 4 categories;

Table 76 API rating for Stage A of recommendation.

Rate	API	AV	BE	C	GA	GR	HE	SE	SA
R	java.applet				X				
R	java.awt.dnd	X							
R	java.awt.font					X			
R	java.awt.print						X		
R	java.lang	X							
R	java.util.jar								X
R	javax.imageio.stream					X			
R	javax.mail			X					
R	javax.servlet		X						
R	javax.servlet.http		X						
R	javax.swing.text.html					X			
R	javax.vecmath					X			
R	org.jdom				X				
R	javax.imageio				X	X			
R	javax.mail.internet			X			X		
R	javax.sound.sampled	X			X				
R	org.apache.commons.logging	X					X		
R	org.apache.log4j	X					X		
R	org.w3c.dom	X						X	
P	java.awt.datatransfer	X				X			X
P	java.nio				X	X			X
P	java.security	X		X					X
P	java.sql	X	X				X		X
P	java.util.logging			X	X		X		X
P	javax.swing.tree	X		X		X		X	
L	java.awt.geom	X			X	X	X	X	
L	javax.xml.parsers		X	X	X			X	X
L	java.awt.image	X		X	X	X	X	X	
L	java.beans	X		X	X	X	X	X	
L	java.util.zip			X	X	X	X	X	X
L	javax.swing.filechooser	X			X	X	X	X	X
L	javax.swing.table	X		X	X		X	X	X
L	javax.swing.text	X		X	X	X	X	X	
L	junit.framework	X		X	X		X	X	X
L	org.xml.sax		X	X	X	X		X	X
F	java.util.regex	X	X	X		X	X	X	X
F	java.awt	X	X	X	X	X	X	X	X
F	java.awt.event	X	X	X	X	X	X	X	X
F	java.io	X	X	X	X	X	X	X	X
F	java.lang.reflect	X	X	X	X	X	X	X	X
F	java.net	X	X	X	X	X	X	X	X
F	java.text	X	X	X	X	X	X	X	X
F	java.util	X	X	X	X	X	X	X	X
F	javax.swing	X	X	X	X	X	X	X	X
F	javax.swing.border	X	X	X	X	X	X	X	X
F	javax.swing.event	X	X	X	X	X	X	X	X

*Audio & Video - AV, Business & Enterprise - BE, Communications - C, Games - G, Graphics - GR, Home & Education - HE, Science & Engineering - SE, and System Administration - SA.

c) Largely recommended (L) represents that an API was recommended in 5 or 6 categories;
d) Fully recommended (F) represents that an API was recommended in 7 or 8 categories.

Thereafter, we counted the occurrence of every API over lists of APIs recommended in tests carried out on Stage A. Then, we used the 4-point rating scale over those lists (Table 76) where we showed rating (rate), 46 APIs, and corresponding category acronym where each API was recommended. We discussed these results in Section 8.9.1.

6.9.2 API recommendation for Stage B

Stage B consists on Software Engineers in advanced stage of software development, i.e., software already uses some APIs. In this Stage, we used the 5-fold cross validation method. Therefore, in every category, for each software from test set, 50% of APIs were removed and saved in an .xml file as the relevant APIs, i.e., APIs that we expected to be recommended for each software. As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration.

In Stage B, we used Collaborative Filtering (CF) recommendation technique along with Frequent Itemset mining (FIS) technique. Both techniques have two main attributes: i) the minimum support threshold value, i.e., *minSupport* in Algorithm 4 (Section 6.3.2.1); and ii) the number of nearest neighbors to consider, i.e., *k* in Algorithm 5 (Section 6.3.2.2). Thus, we needed to find the "best" value for each attribute in every category. In case of *minSupport*, we used the same values found for FIS technique in Stage A, since we used the same data sample, technique, and evaluation method. On the other hand, we used the plug-in and analyzed the effect of varying *k*, i.e., the number of nearest neighbors (NN) in top-10 lists of APIs recommended from all iterations and

replicates of 5-fold cross validation method. As general results, we found that varying *k* in 5 presented best results for Communications, Home & Education, and System Administration categories. Varying *k* in 15 presented best results just for Graphics category. Varying *k* in 20 presented best results just for Audio & Video and Games categories. Varying *k* in 25 presented best results just for Business & Enterprise and Science & Engineering categories.

After using corresponding configuration for every category, we obtained averaged results regarding large lists in Stage B for the eight categories and we found a recall of 52.6%, a precision of 26.6%, and a recall rate of 98.9%. On the other hand, regarding top-20 lists in Stage B for the eight categories, we found a recall of 48.9%, a precision of 29.9%, and a recall rate of 98.3%. We discussed these results in Section 8.9. Furthermore, in order to identify the most and less APIs recommended in Stage B among the eight categories, we established a 4-point rating scale (R, P, L, and F):

a) Rarely recommended (R) represents that an API was recommended in 1 or 2 categories;
b) Partially recommended (P) represents that an API was recommended in 3 or 4 categories;
c) Largely recommended (L) represents that an API was recommended in 5 or 6 categories;
d) Fully recommended (F) represents that an API was recommended in 7 or 8 categories.

Before applying the 4-point rating scale to APIs recommended among the eight categories, we considered that in Stage B, we did five replicates of the recommendation tests for each target software in iterations of the 5-fold cross validation method because the APIs removal was randomized (Sections 5.3 and 5.4). Thus, for counting API occurrence, we selected recommendation lists from the replicate with highest recall value in $N = 20$ variation.

Table 77 API rating for Stage B of recommendation

Rate	API	AV	BE	C	GA	GR	HE	SE	SA
R	java.awt.dnd	X							
R	java.rmi								X
R	java.rmi.registry								X
R	java.rmi.server								X
R	java.security.cert			X					
R	java.security.spec			X					
R	java.util.jar								X
R	java.util.Map						X		
R	java.util.prefs	X							
R	javax.crypto			X					
R	javax.crypto.spec			X					
R	javax.imageio.stream					X			
R	javax.mail.internet			X					
R	javax.management								X
R	javax.naming								X
R	javax.net.ssl			X					
R	javax.print						X		
R	javax.print.attribute						X		
R	javax.print.attribute.standard						X		
R	javax.servlet						X		
R	javax.servlet.http						X		
R	javax.servlet.jsp						X		
R	javax.servlet.jsp.tagext						X		
R	javax.sql						X		
R	javax.swing.plaf.basic			X					
R	javax.swing.plaf.metal			X					
R	javax.swing.text.html					X			
R	javax.vecmath					X			
R	javax.xml.transform.dom								X
R	org.apache.struts						X		
R	org.apache.struts.action		X						
R	org.apache.struts.util						X		
R	org.apache.struts.validator						X		
R	org.apache.xerces.parsers			X					
R	org.jdom.input				X				
R	org.lwjgl.opengl.glu					X			
R	org.lwjgl.util.vector					X			
R	org.xml.sax.helpers			X					
R	javax.activation			X			X		
R	javax.mail			X			X		
R	org.jdom				X				

*Audio & Video - AV, Business & Enterprise - BE, Communications - C, Games - G, Graphics - GR, Home & Education - HE, Science & Engineering - SE, and System Administration - SA.

Then, we counted occurrence of every API over lists of APIs recommended. We used the 4-point rating scale over those lists (Table 77) where we showed rate, 82 APIs, and corresponding category acronym where each API was recommended. We discussed results in Section 8.9.2.

Table 77 API rating for Stage B of recommendation (cont.)

Rate	API	AV	BE	C	GA	GR	HE	SE	SA
P	java.awt.print					X	X	X	
P	java.lang	X						X	X
P	javax.imageio				X	X		X	
P	javax.sound.sampled	X		X	X				
P	javax.swing.plaf					X	X	X	
P	javax.xml.transform	X						X	X
P	javax.xml.transform.stream	X					X		X
P	org.apache.commons.logging			X			X	X	
P	java.applet			X	X	X		X	
P	java.awt.font	X			X	X		X	
P	java.nio.channels	X		X		X			X
P	java.security	X		X	X				X
L	java.awt.datatransfer	X		X		X		X	X
L	java.awt.geom	X			X	X	X	X	
L	java.util.logging			X	X		X	X	X
L	org.w3c.dom	X		X	X			X	X
L	java.awt.image	X		X	X	X	X	X	
L	java.math	X		X		X	X	X	X
L	java.nio	X		X	X	X		X	X
L	javax.swing.tree	X		X	X	X		X	X
L	javax.xml.parsers	X		X	X			X	X
L	junit.framework	X		X	X		X	X	X
L	org.apache.log4j	X		X	X	X	X	X	
L	org.xml.sax	X		X	X	X		X	X
F	java.beans	X		X	X	X	X	X	X
F	java.sql	X	X	X	X		X	X	X
F	java.util.regex	X		X	X	X	X	X	X
F	java.util.zip	X		X	X	X	X	X	X
F	javax.swing.event	X		X	X	X	X	X	X
F	javax.swing.filechooser	X		X	X	X	X	X	X
F	javax.swing.table	X		X	X	X	X	X	X
F	javax.swing.text	X		X	X	X	X	X	X
F	java.awt	X	X	X	X	X	X	X	X
F	java.awt.event	X	X	X	X	X	X	X	X
F	java.io	X	X	X	X	X	X	X	X
F	java.lang.reflect	X	X	X	X	X	X	X	X
F	java.net	X	X	X	X	X	X	X	X
F	java.text	X	X	X	X	X	X	X	X
F	java.util	X	X	X	X	X	X	X	X
F	javax.swing	X	X	X	X	X	X	X	X
F	javax.swing.border	X	X	X	X	X	X	X	X

*Audio & Video - AV, Business & Enterprise - BE, Communications - C, Games - G, Graphics - GR, Home & Education - HE, Science & Engineering - SE, and System Administration - SA.

7 QUANTITATIVE DISCUSSION

Our study showed that our methodology could make useful API recommendations, even in small top-*N* lists of APIs recommended for Software Engineers whose software was categorized and in initial or advanced stage of software development. In this chapter, we discussed the results of every category, considering API recommendation for Stage A and Stage B. Besides, we discussed overall results regarding our study findings and their implications.

The remainder of this Chapter is organized as follows. Section 8.1 discusses results for category Audio & Video. Section 8.2 discusses results for category Business & Enterprise category. Section 8.3 discusses results for Communications category. Section 8.4 discusses results for Games category. Section 8.5 discusses results for Graphics category. Section 8.6 discusses results for Home & Education category. Section 8.7 discusses results for Science & Engineering category. Section 8.8 discusses results for System Administration category. Section 8.9 presents a general discussion.

7.1 Audio & Video category

7.1.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists of APIs recommended (Table 6), we expected maximum recall and low precision values since many irrelevant items could be recommended. As expected, we obtained precision value of 30.6% and recall value of 61.1%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 13). We inspected

the number of APIs recommended in large lists from iterations 1 to 5 and 47, 48, 47, 44, and 32 APIs were recommended respectively. Therefore, those values explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 5 to 13 APIs were expected to be recommended; instead, 47 APIs were recommended where in averaged 67.1% of them appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 11.5%. In case of iteration 5, 33 to 106 APIs were expected to be recommended; instead, 32 APIs were recommended. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (37.1%) and consequently higher precision (55.8%). Furthermore, when we requested for large lists of APIs, regarding all iterations, our recommendation methodology could correctly recommend at least one relevant API for all the requests (recall rate of 100%).

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 7), our recommendation methodology was able to put relevant APIs even in high-ranking positions. For instance, when we requested recommendation lists with 1 and 3 APIs, our methodology could correctly recommend at least one relevant API for all (100%) of the requests. Moreover, we obtained the same results for larger lists. In addition, in Table 7, we also observed that recall values increased along with *N* (i.e., 6.2% to 49.0%) and oppositely precision values decreased (i.e., 100% to 49.4%). In small *N* values, these recall behaviors are normal since for target software cannot be expected to receive all relevant APIs, i.e., from software #20 to software #35 (Table 4), we could not receive all relevant APIs even in largest *N* of 20 since relevant APIs are greater than 20. On the other hand, in small *N* values, these precision behaviors are normal since less irrelevant APIs are expected to be recommended.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 14), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, in all iterations and all of their tests, our methodology recommended a relevant API (100% of precision). On the other hand, in top-20 lists of APIs recommended, precision values tended to get lower in all iterations, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values. For example, recommendation lists for N = 20 for software in iteration 1 where number of APIs vary from 5 to 13.

Regarding recall metric (Figure 15), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended the highest recall values were achieved in all iterations. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for N = 20. Then, that difference is normal since software in iteration 5 used from 32 to 106 relevant APIs. Therefore, we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, recall values were low in all iterations (i.e., 2.3% to 6.7%); again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 16), our recommendation methodology could correctly recommend at least one relevant API for all the requests (100%) in all iterations, even when we requested for small lists.

7.1.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists (Table 10), we expected low precision and maximum recall values since many irrelevant

items could be recommended. As expected, we obtained precision value equal to 23.2% and recall value equal to 60.1%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 98.9% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 17). As we did five replicates for every iteration, we looked for the best replicate in each iteration and we manually inspected the number of APIs recommended in large lists. As exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of APIs from every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1, there were 2 to 6 relevant APIs and in replicate 4, from 34 to 39 APIs were recommended. In iteration 2, there were 7 to 16 relevant APIs and in replicate 3, from 31 to 34 APIs were recommended. In iteration 3, there were 8 to 11 relevant APIs and in replicate 5, from 30 to 36 APIs were recommended. In iteration 4, there were 11 to 15 relevant APIs and in replicate 3, from 26 to 32 APIs were recommended. Finally, in iteration 5, there were 16 to 53 relevant APIs and in replicate 1, from 17 to 30 APIs were recommended.

Regarding large lists of APIs recommended (Figure 17), the data above explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged, 65.1% of the relevant appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were also recommended causing the averaged precision value of 7.9%. On the other hand, in iteration 5, in most cases, the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (38.8%) and consequently higher precision (43.0%). Furthermore, when we

requested for large lists of APIs, regarding iteration 1, our recommendation methodology could correctly recommended at least one relevant API for 94.3% of the requests. For the remaining iterations, our methodology could correctly recommend at least one relevant API for all the requests (100%).

On the other hand, analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 11), our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested recommendation lists with one API, i.e., $N = 1$ or $N = 3$, our methodology could correctly recommend at least one relevant API for 93.1% and 95.4% of the requests correspondingly. Moreover, when we requested recommendation lists with size from 7 to 20, our recommendation methodology could correctly recommend at least one relevant API for 97.7% of the requests. In addition, we also observed that recall values increased along with *N* and oppositely precision values decreased. In small *N* values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as *N* value incremented, precision tended to be lower (i.e., 93.1% to 28.9%) and recall to be higher (i.e., 10.6% to 53.3%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 18), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, in all request tests of iteration 4, our methodology recommended a relevant API (precision of 100%). In addition, in iteration 1, our methodology recommended a relevant API with a precision of 82.9%, and on the remaining iterations (2, 3, and 5) just in a few recommendation request tests our methodology recommended some irrelevant APIs causing precision values of 94.3%. In case of top-20 lists of APIs recommended, precision values tended to get lower in all iterations, especially when just a few relevant APIs are expected to be received in the recommendation lists because more

irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$ with a precision of 13.1%.

Regarding recall metric (Figure 19), we observed how these values increased as N increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since in software of iteration 5 there were from 16 to 53 relevant APIs and we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 4.1% to 19.5%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 20), in iteration 1 even when we requested for small lists of APIs, e.g., $N = 1$ and $N = 3$, our recommendation methodology could correctly recommend at least one relevant API for 82.9%. On the other hand, for the remaining iterations, when we requested small API recommendation lists, e.g., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API above 94.3% of the requests. Furthermore, we observed that from $N = 7$ to $N = 20$, our recommendation methodology correctly recommended at least one relevant API for all the request (100%) in all iterations excepting iteration 1, where correctly recommended at least one relevant API just for 88.6% of the requests.

7.2 Business & Enterprise category

7.2.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists of APIs recommended (Table 15), we expected maximum recall and low precision values since many irrelevant items could be recommended; instead, we obtained precision value of 59.5% and recall value of 27.8%. However, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests. Main reason for those results is the high *minSupport* value that we chose for evaluating our recommendation methodology when using Frequent Itemset mining technique (FIS) in API recommendation for Stage A (Section 7.2.1). Thereby, an API was considered frequent in the Business & Enterprise category if it appeared in at least 40% of the training set of software (i.e., minSupport $\geq$ 0.4). As possible consequences, we cannot expect higher recall than precision values because many relevant APIs could not be part of that 40% and could not be recommended. In addition, the probability of recommending many irrelevant APIs would decrease, causing higher precision.

We discussed in detail results of evaluation metrics for every iteration of the 5-fold cross validation method regarding large lists of APIs recommended (Figure 22). Therefore, we manually inspected the number of APIs recommended in large lists. From iterations 1 to 5, we found that 12, 13, 12, 10, and 9 APIs were recommended respectively. Thus, those values explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 9 to 12 APIs were expected to be recommended and 12 APIs were recommended where in averaged 43.7% of the expected appeared in large lists of APIs recommended. Besides, some irrelevant APIs were recommended and in averaged precision was

36.9%. On the other hand, in iteration 5, 53 to 181 APIs were expected to be recommended; instead, 9 APIs were recommended. Because of that, for target software would not be possible to receive all expected APIs, causing low recall value (9.3%) and consequently higher precision (85.7%). Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for all the requests over the five iterations (i.e., recall rate of 100%).

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 16); we found that our recommendation methodology was able to put relevant APIs even in high-ranking positions. For instance, when we requested for top-*N* lists of APIs in $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for all requests (100% of recall rate). Moreover, we obtained the same results even for larger top-*N* lists. In Table 16, we observed that recall values increased along with *N* (i.e., 7.5% to 27.8%) and oppositely precision values decreased (i.e., 93.4% to 59.5%). In addition, an interesting fact is that both metrics (recall and precision) achieved their maximum or minimum values in top-13, i.e., they stop increasing or decreasing since lists of APIs recommended barely achieved at most a size of 13 APIs, even when requesting top-20 lists of APIs. Once again, that fact is consequence of the higher *minSupport* value selected.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 23), major of these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended regarding tests from all iterations, our methodology could recommended a relevant API (100% of precision). On the other hand, in top-20 lists of APIs recommended, precision values tended to get lower in all iterations, especially when just a few relevant APIs are expected to be received in the lists of APIs recommended because more

irrelevant APIs can appear causing those low precision values. For example, in iteration 1, we obtained the lowest precision value (36.9%) for top-*N* lists of APIs recommended when varying *N* from 13 to 20. An exceptional case of precision was presented in iteration 5 top-*N* lists of APIs recommended at $N = 20$, where we obtained precision value of 85.7%. That case can be explained by the number of APIs used by software in that iteration (from 53 to 181). Thus, as there were many relevant APIs expected, the change of recommending irrelevant APIs decreased.

Regarding recall metric (Figure 24), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. E.g., for all iterations when varying *N* from 13 to 20 in top-*N* lists of APIs recommended, the highest recall values were achieved. However, iterations 4 and 5 presented significant differences regarding the other iterations with top-*N* lists of APIs recommended. Then, those differences are normal since software in iterations 4 and 5 used from 32 to 181 relevant APIs. Therefore, we cannot expect to receive all relevant APIs when just few of them were requested. Instead, in top-1 lists of APIs recommended, recall values were low in all iterations (i.e., 1.3% to 9.8%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 25), even when we requested for small lists of APIs recommended, our recommendation methodology could correctly recommend at least one relevant API for all the requests in all iterations (i.e., recall rate of 100%).

7.2.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists of APIs recommended (Table 19), we expected maximum recall and low precision values

since many irrelevant items could be recommended; Instead, we obtained precision value of 50.2% and recall value of 25.6%. However, when we requested for large lists of APIs, our methodology could correctly recommend at least one relevant API for 97.1% of the requests. Main reason for those results is the high *minSupport* value that we chose for evaluating our recommendation methodology when using Frequent Itemset mining technique (FIS) in API recommendation for Stage B (Section 7.2.2). Thereby, using FIS, an API was considered frequent in the Business & Enterprise category if it appeared in at least 40% of the training set of software (i.e., minSupport $\geq$ 0.4). As possible consequences, we cannot expect higher recall than precision values because many relevant APIs could not be part of that 40% and could not be recommended. In addition, the probability of recommending many irrelevant APIs would decrease, causing higher precision.

We discussed in detail results of evaluation metrics for every iteration of the 5-fold cross validation method regarding large lists of APIs recommended (Figure 26). As we did five replicates for every iteration, we looked for the best replicate in each iteration and we manually inspected the number of APIs recommended in large lists. As exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of APIs from every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1, there were 4 to 6 relevant APIs, and in replicate 1, from 8 to 11 APIs were recommended. In iteration 2, there were 7 to 9 relevant APIs and in replicate 2, from 6 to 10 APIs were recommended. In iteration 3, there were 9 to 14 relevant APIs, and in replicate 2, from 6 to 10 APIs were recommended. In iteration 4, there were 16 to 26 relevant APIs and in replicate 1, from 3 to 7 APIs were recommended. Finally, in iteration 5, there were 26 to 90 relevant APIs and in replicate 5, from 4 to 8 APIs were recommended.

Regarding large lists of APIs recommended (Figure 26), the data above explain why recall tended to get lower through iterations, and as consequence of

the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged, 45.8% of the relevant appeared in lists of APIs recommended with a precision of 27.6%. On the other hand, in iteration 5, in most cases, the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, for target software would not be possible to receive all expected APIs, causing low recall value (9.1%) and consequently higher precision value (78.9%). Furthermore, when we requested for large lists of APIs regarding iteration 1 and 4, our recommendation methodology could correctly recommended at least one relevant API for 94.3% and 91.4% of the requests respectively. For the remaining iterations, our methodology could correctly recommend at least one relevant API for all the requests (i.e., recall rate of 100%).

On the other hand, analyzing our recommendation regarding small lists, i.e., variations of *N* in top-*N* lists of APIs recommended (Table 20); we found that our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested for top-*N* lists of APIs at $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for 87.4% and 92.0% of the requests respectively. Moreover, when we requested for top-*N* lists of APIs with *N* from 10 to 20, our recommendation methodology could correctly recommend at least one relevant API for 97.1% of the requests. In addition, recall values increased along with *N* and oppositely precision values decreased. In small *N* values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as *N* value incremented, precision tended to be lower (i.e., 87.6% to 50.2%) and recall to be higher (i.e., 7.9% to 25.6%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 27), these values decremented as *N* incremented in all iterations. For

example, in top-1 lists of APIs recommended, in almost all request tests in iteration 5, our methodology recommended a relevant API (precision of 94.3%). In addition, for remaining iterations, our methodology recommended a relevant API in a top-1 for more than 77.9% of the requests. In case of top-20 lists of APIs recommended, in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in lists of APIs recommended because more irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$ with a precision of 27.6%. An exceptional case of precision was presented in iteration 5 at $N = 20$, where we obtained the highest precision value of 78.9%. That case can be explained by the number of relevant APIs used by software in that iteration (from 26 to 90). Thus, as there were used many relevant APIs, change of recommending irrelevant APIs decreased, causing those precision values.

Regarding recall metric (Figure 28), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. However, in all iterations, varying *N* from 13 to 20 in top-*N* lists did not represent any improvement since maximum recall values were already achieved. On the other hand, in all iterations of top-1 lists of APIs recommended, recall values were low (i.e., 2.4% to 16.1%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 29), in iteration 5, even when we requested for small lists of APIs, e.g., $N = 1$ and $N = 3$, our recommendation methodology could correctly recommend at least one relevant API for 94.3% of the requests. For remaining iterations at same *N* values of top-*N* lists, our recommendation methodology could correctly recommend at least one relevant API more than 85,7% of the requests. Furthermore, in iterations 2, 4, and 5 our

methodology could correctly recommend at least one relevant API for all of the requests (i.e., recall rate of 100%) for top-*N* lists in *N* values from 7 to 20.

7.3 Communications category

7.3.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists (Table 24), we expect maximum recall and low precision values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 26.4% and a recall value of 58.2%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 98.0% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 31). We inspected the number of APIs recommended in large lists from iterations 1 to 5 and 57, 53, 54, 42, and 37 APIs were recommended respectively. Therefore, those values explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 7 to 12 APIs were expected to be recommended; instead, 57 APIs were recommended where in averaged 71.2% of them appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 11.2%. Instead, in iteration 5, 42 to 65 APIs were expected to be recommended; instead, 37 APIs were recommended. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (39.5%) and consequently higher precision (52.2%). Furthermore, when we requested for large lists of APIs, regarding iteration 1, our recommendation methodology could correctly recommend at least one relevant API for 90.0% of the requests. For the remaining

iterations, our methodology could correctly recommend at least one relevant API for 100% of the requests.

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 25), our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested recommendation lists with $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for 96.0% of the requests. Moreover, when we requested recommendation lists with size from 5 to 20 APIs, our recommendation methodology could correctly recommend at least one relevant API for 98.0% of the requests. In addition, we also observed that recall values increased along with *N* and oppositely precision values decreased (Table 25). In small *N* values, these behaviors are expected since we cannot expect to receive all relevant APIs when just few had been requested. Thus, as *N* value incremented, precision tended to be lower (i.e., 96.0% to 41.3%) and recall to be higher (5.5% to 41.2%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 32), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, from iteration 2 to 5 in all tests, our methodology recommended a relevant API (100% of precision) and just in a few recommendation request tests of iteration 1, our methodology recommended an irrelevant API causing a precision of 80%. On the other hand, in top-20 lists of APIs recommended, precision values tended to get lower in all iterations, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$ with 22.5% of precision. An exceptional case of precision was presented in iteration 5, where we obtained the highest precision value of 65.0% for recommendation lists at $N = 20$. That case

can be explained by the number of APIs used by software in that iteration (from 34 to 65). Thus, as there were used many APIs, the change of recommending irrelevant APIs decreased, causing that behavior in precision values.

Regarding recall metric (Figure 33), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iterations 4 and 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Those differences are normal since software in iterations 4 and 5 have more than 20 relevant APIs, and we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 2.1% to 8.8%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 34), in iteration 1 even when we requested for small lists of APIs, e.g., $N = 3$ and $N = 5$, our recommendation methodology could correctly recommend at least one relevant API for 80.0% of the requests. On the other hand, for the remaining iterations when varying the size of recommendation lists, either large or small, our methodology could correctly recommend at least one relevant API for 100% of the requests.

7.3.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists (Table 28), we expected low precision and maximum recall values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 26.3% and recall value equal to 48.2%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 98.0% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 35). As we did five replicates for every iteration, we looked for the best replicate in each iteration and we manually inspected the number of APIs recommended in large lists. As exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of the APIs from every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1, there were 3 to 6 relevant APIs and in replicate 3, from 20 to 32 APIs were recommended. In iteration 2, there were 6 to 8 relevant APIs and in replicate 4, from 17 to 27 APIs were recommended. In iteration 3, there were 8 to 10 relevant APIs and in replicate 1, from 17 to 24 APIs were recommended. In iteration 4, there were 10 to 15 relevant APIs and in replicate 5, from 9 to 24 APIs were recommended. Finally, in iteration 5, there were 17 to 32 relevant APIs and in replicate 1, from 14 to 24 APIs were recommended.

Regarding large lists of APIs recommended (Figure 35), the data above explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged 61.2% of relevant APIs appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were also recommended causing the averaged precision value of 11.2%. On the other hand, in iteration 5, in most cases, the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (39.5%) and consequently higher precision (52.2%). Furthermore, when we requested for large lists of APIs, regarding iteration 1, our recommendation methodology could correctly recommended at least one relevant API for 90.0% of the requests. For the remaining iterations, our methodology could correctly recommend at least one relevant API for 100% of the requests.

On the other hand, analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 29), our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested recommendation lists with one API, i.e., $N = 1$, our methodology could correctly recommend at least one relevant API for 92.8% of the requests. Moreover, when we requested recommendation lists with size from 10 to 20, our recommendation methodology could correctly recommend at least one relevant API for 97.6% of the requests. In addition, in Table 29, we also observed that recall values increased along with *N* and oppositely precision values decreased. In small *N* values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as *N* value incremented, precision tended to be lower (i.e., 92.8% to 27.6%) and recall to be higher (i.e., 10.1% to 46.6%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 36), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, in all request tests of iterations 3 and 5, our methodology recommended a relevant API (precision of 100%) and on the remaining iterations (1, 2, and 4) just in a few recommendation request tests our methodology recommended some irrelevant APIs causing precision values of 72.0%, 98.0 and 94.0% correspondingly. In case of top-20 lists of APIs recommended in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$.

Regarding recall metric (Figure 37), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For

example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iterations 4 and 5 presented significant difference regarding the other iterations with recommendation lists for N = 20. Those differences are normal since software in iterations 4 and 5 there were more than 20 relevant APIs, and we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 4.1% to 15.2%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 38), in iteration 1 even when we requested for small lists of APIs, e.g., N = 1 and N = 3, our recommendation methodology could correctly recommend at least one relevant API for 72.0% and 78% of the requests correspondingly. On the other hand, for the remaining iterations, when we requested small API recommendation lists, e.g., N = 1 and N = 3, our methodology could correctly recommend at least one relevant API above 94% of the requests. Furthermore, we observed that from N = 10 to N = 20, our recommendation methodology correctly recommended at least one relevant API for 100.0% of the requests in all iterations excepting iteration 1, where correctly recommended at least one relevant API just for 88.0% of the requests.

7.4 Games category

7.4.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists (Table 33), we expect maximum recall and low precision values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 33.7% and a recall value of 66.0%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 40). We inspected the number of APIs recommended in large lists from iterations 1 to 5 and 41, 35, 35, 33, and 26 APIs were recommended respectively. Therefore, those values explain why recall tended to get lower through iterations, as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 5 to 8 APIs were expected to be recommended; instead, 41 APIs were recommended where in averaged 87.9% of them appeared in the lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 10.9%. In iteration 5, 26 to 108 APIs were expected to be recommended; instead, 26 APIs were recommended. Because of that, for target software would not be possible to receive all the expected APIs in all tests, causing low recall value (43.8%) and consequently higher precision (42.0%). Furthermore, when we requested for large lists of APIs regarding iterations 1 and 3, our recommendation methodology could correctly recommend at least one relevant API for 98.6% of the requests. For the remaining iterations, our methodology could correctly recommend at least one relevant API for 100% of the requests.

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 34), our recommendation methodology was able to put relevant APIs even in high-ranking positions. For instance, when we requested recommendation lists with 1 or 3 APIs, i.e., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for 98.6 and 100% of the requests correspondingly. Moreover, when we requested recommendation lists with larger sizes, i.e., from $N = 5$ to $N = 20$, our methodology could correctly recommend at least one relevant API for all the request tests (recall rate of 100%). In addition, in Table 34, we also observed that recall values increased along with N (i.e., 7.3% to 57.3%) and oppositely precision

values decreased (i.e., 98.6% to 45.4%). In small *N* values, these recall behaviors are normal since for target software cannot be expected to receive all relevant APIs, i.e., from software #47 to software #70 (Table 31) we could not receive all relevant APIs even in largest *N* of 20 since relevant APIs are greater than 20. On the other hand, in small *N* values, these precision behaviors are normal since less irrelevant APIs are expected to be recommended.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 41), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, in all iterations we obtained precision values above 84.0%. On the other hand, in top-20 lists of APIs recommended, in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values. For example, recommendation lists for $N = 20$ for software in iteration 1 where number of APIs varied from 5 to 8.

Regarding recall metric (Figure 42), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since software in iteration 5 used from 26 to 108 relevant APIs. Therefore, we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 5.2% to 23.9%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 42), in all iterations, when we requested for small lists of APIs, e.g., $N = 1$ and $N = 3$ our recommendation methodology could correctly recommend at least one relevant API for more than 84.0% of the requests. In addition, from $N = 5$, our recommendation methodology could correctly recommend at least one relevant API to more request tests, i.e., to more than 98.6% of the requests, even achieving 100% in some cases, like iterations 5 and 6.

7.4.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists (Table 37), we expected low precision and maximum recall values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 24.1% and recall value equal to 64.3%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 99.4% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 44). As we did five replicates for every iteration, we looked for the best replicate in each iteration and we manually inspected the number of APIs recommended in large lists. As exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of the APIs from every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1, there were 2 to 4 relevant APIs and in replicate 3, from 27 to 33 APIs were recommended. In iteration 2, there were 4 to 7 relevant APIs and in replicate 4, from 24 to 32 APIs were recommended. In iteration 3, there were 7 to 9 relevant APIs and replicate in replicate 5, from 17 to 32 APIs were recommended. In iteration 4, there were 10 to 12 relevant APIs and in replicate 2, from 18 to 27

APIs were recommended. Finally, in iteration 5, there were 13 to 54 relevant APIs and in replicate 2, from 16 to 24 APIs were recommended.

Regarding large lists of APIs recommended (Figure 44), the data above explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged, 89.6% of the relevant appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were also recommended causing the averaged precision value of 14.8%. On the other hand, in iteration 5, in most cases, the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (42.1%) and consequently higher precision (58.5%). Furthermore, when we requested for large lists of APIs, regarding all iterations, our methodology could correctly recommend at least one relevant API for all the requests (100%).

On the other hand, analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 38), our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested recommendation lists with one API, i.e., $N = 1$ or $N = 3$, our methodology could correctly recommend at least one relevant API for 89.4% and 97.4% of the requests correspondingly. Moreover, when we requested recommendation lists with size from 7 to 20, our recommendation methodology could correctly recommend at least one relevant API for more than 99.0% of the requests. In addition, we also observed that recall values increased along with N and oppositely precision values decreased. In small N values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as N value incremented, precision tended to be lower (i.e., 89.4% to 26.7%) and recall to be higher (i.e., 12.5% to 61.6%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 45), these values decremented as *N* incremented in all iterations. For example, in top-1 of recommendation of iteration 3, just in a few recommendation request tests our methodology recommended some irrelevant APIs causing a precision value of 92.9%. In addition, for the remaining iterations, our methodology recommended a relevant API with a precision of 100. In case of top-20 lists of APIs recommended in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$ with a precision of 29.3%.

Regarding recall metric (Figure 46), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since in software of iteration 5 there were from 13 to 54 relevant APIs and we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 2.8% to 15.1%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 20), in iteration 3 even when we requested for small lists of APIs, e.g., $N = 1$, our recommendation methodology could correctly recommend at least one relevant API for 92.9%. On the other hand, for the remaining iterations, when we requested small API recommendation lists, e.g., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for 100% of the requests. Furthermore, we observed that from *N*

= 5 to N = 20, our recommendation methodology correctly recommended at least one relevant API for all the request (100%) in all iterations.

7.5 Graphics category

7.5.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists (Table 42), we expect maximum recall and low precision values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 26.0% and a recall value of 65.4%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 49). We inspected the number of APIs recommended in large lists from iterations 1 to 5, we found that 51, 47, 48, 37, and 44 APIs were recommended respectively. Therefore, those values explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 8 to 12 APIs were expected to be recommended; instead, 51 APIs were recommended where in averaged 90.8% of them appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 17.6%. Instead, in iteration 5, 25 to 42 APIs were expected to be recommended; instead, 44 APIs were recommended where in averaged 54.3% of them appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 35.8%. Furthermore, when we requested for large lists of APIs, regarding all iterations, our recommendation

methodology could correctly recommend at least one relevant API for all the requests (recall rate of 100%).

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 43), our recommendation methodology was able to put relevant APIs even in high-ranking positions. For instance, when we requested recommendation lists with 1 or 3 APIs, i.e., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for all requests (recall rate of 100%). Moreover, we obtained the same results for larger recommendation lists. In addition, in Table 43, we also observed that recall values increased along with *N* (i.e., 6.2% to 52.7%) and oppositely precision values decreased (i.e., 100% to 46.3%). In small *N* values, these recall behaviors are normal since for target software cannot be expected to receive all relevant APIs, i.e., from software #13 to software #20 (Table 4) we could not receive all relevant APIs even in largest *N* of 20 since relevant APIs are greater than 20. On the other hand, in small *N* values, these precision behaviors are normal since less irrelevant APIs are expected to be recommended.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 50), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, in all iterations and in all of their tests, our methodology recommended a relevant API (100% of precision). On the other hand, in top-20 lists of APIs recommended, in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values. For example, recommendation lists for $N = 20$ for software in iteration 2 where number of APIs varied from 12 to 16.

Regarding recall metric (Figure 51), we observed how these values increased as *N* increased. That result is normal and expected, since between more

APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 1 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since software in iteration 1 used from 8 to 12 relevant APIs. Therefore, we there is more change to obtain those APIs in top-20 lists. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 3.5% to 10.3%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 52), our recommendation methodology could correctly recommend at least one relevant API for all the requests (100%) in all iterations, even when we requested for small lists.

7.5.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists (Table 46), we expected low precision and maximum recall values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 15.0% and recall value equal to 66.7%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 53). As we did five replicates for every iteration, we looked for the best replicate in each iteration and we manually inspected the number of APIs recommended in large lists. As exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of the APIs from every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1, there were 4 to 6 relevant APIs and replicate 5, from 46 to 48 APIs

were recommended. In iteration 2, there were 12 to 16 relevant APIs and in replicate 3, from 43 to 49 APIs were recommended. In iteration 3, there were 9 relevant APIs and in replicate 5 from 43 to 45 APIs were recommended. In iteration 4, there were 10 to 12 relevant APIs and in replicate 2, from 29 to 36 APIs were recommended. Finally, in iteration 5, there were from 12 to 21 relevant APIs and in replicate 3, from 38 to 40 APIs were recommended.

Regarding large lists of APIs recommended (Figure 53), the data above explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged, 92.3% of the relevant appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were also recommended causing the averaged precision value of 10.0%. On the other hand, in iteration 5, just in one case the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, just in one case would not be possible to receive all the expected APIs. But as most cases the number of APIs recommended was greater than the number of relevant APIs, it caused an averaged recall value of 56.0% and an averaged precision value of 21.3%. Furthermore, when we requested for large lists of APIs, regarding all iterations, our methodology could correctly recommend at least one relevant API for all the requests (100%).

On the other hand, analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 47), our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested recommendation lists with one API, i.e., $N = 1$ or $N = 3$, our methodology could correctly recommend at least one relevant API for 98.0% and 100% of the requests correspondingly. Moreover, when we requested recommendation lists with size from 5 to 20, our recommendation methodology could correctly recommend at least one relevant

API for 100% of the requests. In addition, we also observed that recall values increased along with *N* and oppositely precision values decreased. In small *N* values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as *N* value incremented, precision tended to be lower (i.e., 98.0% to 25.0%) and recall to be higher (i.e., 11.9% to 55.8%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 54), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, in almost all request tests of iteration 3, our methodology recommended a relevant API (precision of 90.0%). For the remaining iterations, in all request tests, our methodology recommended a relevant API (precision of 100%). In case of top-20 lists of APIs recommended in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values, e.g., iteration 2 at $N = 20$ with a precision of 17.5%.

Regarding recall metric (Figure 55), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 1 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since in software of iteration 1 there were from 4 to 6 and there is more probability of recommending then in a top-20. Even so, it did not recommend all of them, and instead achieved 81.3% of them. On the other hand, in top-1 of recommendation, in all iterations recall values were low (i.e., 6.7% to 19.6%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 56), in iteration 3, when we requested for small lists of APIs, e.g., $N = 1$, our recommendation methodology could correctly recommend at least one relevant API for 90.0%. On the other hand, for the remaining iterations, when we requested small API recommendation lists, e.g., $N = 1$, our methodology could correctly recommend at least one relevant API for 100% of the requests. Furthermore, we observed that from $N = 3$ to $N = 20$, our recommendation methodology correctly recommended at least one relevant API for all the request (recall rate of 100%).

7.6 Home & Education category

7.6.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists (Table 51), we expect maximum recall and low precision values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 22.4% and a recall value of 57.0%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 58). We inspected the number of APIs recommended in large lists from iterations 1 to 5, we found that 50, 51, 48, 37, and 26 APIs were recommended respectively. Therefore, those values explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 5 to 9 APIs were expected to be recommended; instead, 50 APIs were recommended where in averaged 87.7% of them appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 12.5%. Instead,

in iteration 5, 19 to 37 APIs were expected to be recommended; instead, 26 APIs were recommended. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (22.7%) and consequently higher precision (40.4%). Furthermore, when we requested for large lists of APIs, regarding all iterations, our recommendation methodology could correctly recommend at least one relevant API for all the requests (recall rate of 100%).

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 52), our recommendation methodology was able to put relevant APIs even in high-ranking positions. For instance, when we requested recommendation lists with 1 or 3 APIs, i.e., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for all (100%) of the requests. Moreover, we obtained the same results for larger recommendation lists. In addition, in Table 52, we also observed that recall values increased along with N (i.e., 7.7% to 53.3%) and oppositely precision values decreased (i.e., 100% to 39.0%). In small N values, these recall behaviors are normal since for target software cannot be expected to receive all relevant APIs, i.e., from software #15 to software #20 (Table 49) we could not receive all relevant APIs even in largest N of 20 since relevant APIs are greater than 20. On the other hand, in small N values, these precision behaviors are normal since less irrelevant APIs are expected to be recommended.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 59), these values decremented as N incremented in all iterations. For example, in top-1 lists of APIs recommended, in all iterations and in all of their tests, our methodology recommended a relevant API (100% of precision). On the other hand, in top-20 lists of APIs recommended, in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear

causing those low precision values. For example, recommendation lists for N = 20 for software in iteration 1 where number of APIs vary from 5 to 9.

Regarding recall metric (Figure 60), we observed how these values increased as N increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for N = 20. Then, that difference is normal since software in iteration 5 used from 38 to 75 relevant APIs. Therefore, we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 2.2% to 14.5%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 61), our recommendation methodology could correctly recommend at least one relevant API for all the requests (100%) in all iterations, even when we requested for small lists.

7.6.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists (Table 55), we expected low precision and maximum recall values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 26.8% and recall value equal to 50.0%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 62). As we did five replicates for every iteration, we looked for the best replicate in each iteration and we manually inspected the number of APIs recommended in large lists. As

exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of the APIs from every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1, there were 2 to 4 relevant APIs and in replicate 1 from 17 to 24 APIs were recommended. In iteration 2, there were 5 to 6 relevant APIs and in replicate 4, from 11 to 43 APIs were recommended. In iteration 3, there were 6 to 7 relevant APIs and replicate 3, from 15 to 18 APIs were recommended. In iteration 4, there were 8 to 16 relevant APIs and in replicate 2, from 10 to 30 APIs were recommended. Finally, in iteration 5, there were 19 to 37 relevant APIs and in replicate 1, from 13 to 20 APIs were recommended.

Regarding large lists of APIs recommended (Figure 62), the data above explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged, 84.3% of the relevant appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were also recommended causing the averaged precision value of 18.7%. On the other hand, in iteration 5, in most cases, the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (19.7%) and consequently higher precision (28.2%). Furthermore, when we requested for large lists of APIs, regarding iteration 2, our recommendation methodology could correctly recommended at least one relevant API for 85.0% of the requests. For the remaining iterations, our methodology could correctly recommend at least one relevant API for all the requests (100%).

On the other hand, analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 56), our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested recommendation lists with one API,

i.e., $N = 1$ or $N = 3$, our methodology could correctly recommend at least one relevant API for 87.0% and 92.0% of the requests correspondingly. Moreover, when we requested recommendation lists with size from 7 to 20, our recommendation methodology could correctly recommend at least one relevant API for 97.0% of the requests. In addition, we also observed that recall values increased along with N and oppositely precision values decreased. In small N values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as N value incremented, precision tended to be lower (i.e., 87.0% to 27.5%) and recall to be higher (i.e., 12.7% to 49.8%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 63), these values decremented as N incremented in all iterations. For example, in top-1 lists of APIs recommended, in all request tests of iteration 4, our methodology recommended a relevant API (precision of 100%). On the remaining iterations, just in a few recommendation request tests our methodology recommended some irrelevant APIs causing precision values between 75.0% and 90.0%. In case of top-20 lists of APIs recommended in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$ with a precision of 19.0%.

Regarding recall metric (Figure 64), we observed how these values increased as N increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since in software of iteration 5 there were from 19 to 37

relevant APIs and we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 3.4% to 23.3%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 65), in iteration 4, when we requested for small lists of APIs, e.g., $N = 1$ and $N = 3$, our recommendation methodology could correctly recommend at least one relevant API for 100%. On the other hand, for the remaining iterations, when we requested small API recommendation lists, e.g., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for more than 75.0% of the requests. Furthermore, we observed that from $N = 7$ to $N = 20$, our recommendation methodology correctly recommended at least one relevant API for all the request (100%) in all iterations excepting iteration 2, where correctly recommended at least one relevant API just for 85.0% of the requests.

7.7 Science & Engineering category

7.7.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists (Table 60), we expect maximum recall and low precision values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 28.6% and a recall value of 55.5%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 67). We inspected the number of APIs recommended in large lists from iterations 1 to 5, we found that 46, 47, 44, 43, and 36 APIs were recommended respectively. Therefore, those

values explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 5 to 11 APIs were expected to be recommended; instead, 46 APIs were recommended where in averaged 74.0% of them appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 12.2%. Instead, in iteration 5, 39 to 73 APIs were expected to be recommended; instead, 36 APIs were recommended. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (33.4%) and consequently higher precision (48.1%). Furthermore, when we requested for large lists of APIs, regarding all iterations, our recommendation methodology could correctly recommend at least one relevant API for all the requests (recall rate of 100%).

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 61), our recommendation methodology was able to put relevant APIs even in high-ranking positions. For instance, when we requested recommendation lists with 1 or 3 APIs, i.e., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for 98.0% and 100% of the requests correspondingly. Moreover, our methodology could correctly recommend at least relevant API for all requests in lists with size from 5 to 20. In addition, in Table 61, we also observed that recall values increased along with *N* (i.e., 5.8% to 43.7%) and oppositely precision values decreased (i.e., 98.0% to 45.8%). In small *N* values, these recall behaviors are normal since for target software cannot be expected to receive all relevant APIs, i.e., from software #23 to software #50 (Table 58) we could not receive all relevant APIs even in largest *N* of 20 since relevant APIs are greater than 20. On the other hand, in small *N* values, these precision behaviors are normal since less irrelevant APIs are expected to be recommended.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 68), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, from iteration 2 to 5 in all tests, our methodology recommended a relevant API (100% of precision) and just in a few recommendation request tests of iteration 1, our methodology recommended an irrelevant API causing a precision of 90.0%. On the other hand, in top-20 lists of APIs recommended, in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values. For example, recommendation lists for $N = 20$ for software in iteration 1 where number of APIs varied from 5 to 11.

Regarding recall metric (Figure 69), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since software in iteration 5 used from 39 to 73 relevant APIs. Therefore, we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 2.0% to 11.5%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 70), in iteration 1, even when we requested for small lists of APIs, e.g., $N = 1$, our recommendation methodology could correctly recommend at least one relevant API for 90.0%. On the other hand, for the remaining iterations, when we requested for small or large API

recommendation lists, our methodology could correctly recommend at least one relevant API for 100% of the requests.

7.7.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists (Table 64), we expected low precision and maximum recall values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 19.8% and recall value equal to 52.9%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 98.4% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 71). As we did five replicates for every iteration, we looked for the best replicate in each iteration and we manually inspected the number of APIs recommended in large lists. As exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of the APIs of every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1 there were 2 to 5 relevant APIs and in replicate 2, from 36 to 38 APIs were recommended. In iteration 2, there were 5 to 8 relevant APIs and in replicate 5, from 33 to 39 APIs were recommended. In iteration 3, there were 8 to 11 relevant APIs and replicate 1, from 26 to 34 APIs were recommended. In iteration 4 there were 14 to 19 relevant APIs and in replicate 1, from 27 to 40 APIs were recommended. Finally, in iteration 5, there were 19 to 36 relevant APIs and in replicate 3, from 24 to 32 APIs were recommended.

Regarding large lists of APIs recommended (Figure 71), the data above explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged, 71.9% of the relevant appeared in

lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were also recommended causing the averaged precision value of 7.9%. On the other hand, in iteration 5, in most cases, the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (33.9%) and consequently higher precision (34.0%). Furthermore, when we requested for large lists of APIs, regarding iterations 1 and 2, our recommendation methodology could correctly recommended at least one relevant API for 96.0% of the requests. For the remaining iterations, our methodology could correctly recommend at least one relevant API for all the requests (100%).

On the other hand, analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 65), our recommendation methodology was able to put relevant APIs in high-ranking positions. For instance, when we requested recommendation lists with one API, i.e., $N = 1$ or $N = 3$, our methodology could correctly recommend at least one relevant API for 90.8% and 94.4% of the requests correspondingly. Moreover, when we requested recommendation lists with size from 10 to 20, our recommendation methodology could correctly recommend at least one relevant API for 98.0% of the requests. In addition, we also observed that recall values increased along with N and oppositely precision values decreased. In small N values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as N value incremented, precision tended to be lower (i.e., 90.8% to 25.5%) and recall to be higher (i.e., 10.1% to 46.0%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 72), these values decremented as N incremented in all iterations. For example, in top-1 lists of APIs recommended, in all request tests of iteration 5, our methodology recommended a relevant API (precision of 100%). In addition,

in iterations 1 and 2, our methodology recommended a relevant API with a precision of 84.0%. In iteration 4 and 5 just in a few recommendation request tests our methodology recommended some irrelevant APIs causing precision values of 88.0% and 98.0% correspondingly. In case of top-20 lists of APIs recommended in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$ with a precision of 12.9%.

Regarding recall metric (Figure 73), we observed how these values increased as N increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 1 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since software in iteration 1 used from 2 to 5 relevant APIs. Therefore, there is more change to obtain those APIs in top-20 lists. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 3.9% to 20.1%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 74), in iterations 1 and 2, when we requested for small lists of APIs, e.g., $N = 1$ and $N = 3$, our recommendation methodology could correctly recommend at least one relevant API for 84.0%. On iteration 3 and 4, when we requested small API recommendation lists, e.g., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API above 98.0% and 88.0% of the requests correspondingly. Finally, in iteration 5, either for small or large lists, our recommendation methodology could correctly recommend at least one relevant API to all request tests (100% of recall rate). Furthermore, we observed that from $N = 5$ to $N = 20$, our recommendation

methodology correctly recommended at least one relevant API for more than 90.0% of request in all iterations.

7.8 System Administration category

7.8.1 API recommendation for Stage A

When analyzing our recommendation methodology for large lists (Table 69), we expect maximum recall and low precision values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 23.1% and a recall value of 62.8%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 100% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 76). We inspected the number of APIs recommended in large lists from iterations 1 to 5, we found that 60, 59, 59, 46, and 28 APIs were recommended respectively. Therefore, those values explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between these metrics, precision tended to get higher. For instance, in iteration 1, 5 to 7 APIs were expected to be recommended and instead, 60 APIs were recommended where in averaged 83.7% of them appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were recommended and in averaged precision was 8.6%. Instead, in iteration 5, 25 to 147 APIs were expected to be recommended; instead, 28 APIs were recommended. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (29.4%) and higher precision (54.1%). Furthermore, in large API recommendation lists regarding all iterations, our recommendation methodology could correctly recommend at least one relevant API for all the requests (recall rate of 100%).

On the other hand, when analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 70), our recommendation methodology was able to put relevant APIs even in high-ranking positions. For instance, when we requested recommendation lists with 1 or 3 APIs, i.e., $N = 1$ and $N = 3$, our methodology could correctly recommend at least one relevant API for 97.1% and 100% of the requests correspondingly. Moreover, our methodology could correctly recommend at least one relevant API for 100% of the requests for recommendation lists for the remaining sizes. In addition, in Table 70, we also observed that recall values increased along with *N* (i.e., 7.7% to 57.1%) and oppositely precision values decreased (i.e., 94.3% to 37.9%). In small *N* values, these recall behaviors are normal since for target software cannot be expected to receive all relevant APIs, i.e., from software #25 to software #35 (Table 67) we could not receive all relevant APIs even in largest *N* of 20 since relevant APIs are greater than 20. On the other hand, in small *N* values, these precision behaviors are normal since less irrelevant APIs are expected to be recommended.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 77), these values decremented as *N* incremented in all iterations. For example, in top-1 lists of APIs recommended, iterations 2, 3 and 5 in all of their tests, our methodology recommended a relevant API (100% of precision). For the remaining iterations in some tests were recommended irrelevant APIs causing a precision of 85.7%. Conversely, in top-20 of recommendation, in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values. E.g., recommendation lists at $N = 20$ for software in iteration 1 where number of APIs vary from 5 to 7.

Regarding recall metric (Figure 78), we observed how these values increased as *N* increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that difference is normal since software in iteration 5 used from 25 to 147 relevant APIs. Therefore, we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 2.2% to 14.2%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 79), in iteration 1, even when we requested for small lists of APIs, e.g., $N = 1$, our recommendation methodology could correctly recommend at least one relevant API for 85.7%. On the other hand, for the remaining iterations, when we requested for small or large API recommendation lists, our methodology could correctly recommend at least one relevant API for 100% of the requests.

7.8.2 API recommendation for Stage B

When analyzing our recommendation methodology for large lists (Table 73), we expected low precision and maximum recall values since many irrelevant items could be recommended. As expected, we obtained precision value equal to 27.4% and recall value equal to 52.9%. Furthermore, when we requested for large lists of APIs, our recommendation methodology could correctly recommend at least one relevant API for 99.4% of the requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 80). As we did five replicates for every iteration, we looked for the best replicate in each iteration and

we manually inspected the number of APIs recommended in large lists. As exposed in the methodology for evaluating this stage of recommendation, the plug-in randomly removed half of APIs from every target software, saving them as the relevant APIs, i.e., APIs expected to be recommended. Thus, in iteration 1 there were 2 to 3 relevant APIs and in replicate 1, from 16 to 30 APIs were recommended. In iteration 2, there were 4 to 6 relevant APIs and in replicate 3, from 9 to 31 APIs were recommended. In iteration 3, there were 6 to 8 and in replicate 3, from 9 to 18 APIs were recommended. In iteration 4, there were 9 to 12 relevant APIs and in replicate 1, from 13 to 23 APIs were recommended. Finally, in iteration 5 there were 12 to 73 relevant APIs and in replicate 2, from 10 to 20 APIs were recommended.

Regarding large lists of APIs recommended (Figure 80), the data above explain why recall tended to get lower through iterations, and as consequence of the inversely dependence between precision and recall, precision tended to get higher. For instance, in iteration 1, in averaged, 83.3% of the relevant appeared in lists of APIs recommended. Because of the sizes of lists, more irrelevant APIs were also recommended causing the averaged precision value of 15.0%. On the other hand, in iteration 5, in most cases, the number of the APIs recommended was smaller than the number of relevant APIs. Because of that, for target software would not be possible to receive all the expected APIs, causing low recall value (20.3%) and consequently higher precision (45.0%). Furthermore, when we requested for large lists of APIs, regarding iteration 2, our recommendation methodology could correctly recommended at least one relevant API for 97.1% of the requests. For the remaining iterations, our methodology could correctly recommend at least one relevant API for all the requests (100%).

On the other hand, analyzing our recommendation regarding small lists, i.e., varying N in top-N lists of APIs recommended (Table 74), our recommendation methodology was able to put relevant APIs in high-ranking

positions. For instance, when we requested recommendation lists with one API, i.e., $N = 1$ or $N = 3$, our methodology could correctly recommend at least one relevant API for 89.1% and 97.1% of the requests correspondingly. Moreover, when we requested recommendation lists with size from 13 to 20, our recommendation methodology could correctly recommend at least one relevant API for 99.4% of the requests. In addition, we also observed that recall values increased along with N and oppositely precision values decreased. In small N values, these behaviors are expected since cannot be expected to receive all relevant APIs. Thus, as N value incremented, precision tended to be lower (i.e., 89.1% to 27.9%) and recall to be higher (i.e., 13.4% to 52.7%).

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding small lists of APIs recommended. Regarding precision metric (Figure 81), these values decremented as N incremented in all iterations. For example, in top-1 lists of APIs recommended, in iterations 3 and 5, just in a few recommendation request tests our methodology recommended some irrelevant APIs causing precision values of 97.1% and 94.3% correspondingly. For iterations 1, 2, and 4 were recommended more irrelevant APIs in top-1, causing precision values of 80.0%, 88.6%, and 85.7% correspondingly. On the other hand, in case of top-20 lists of recommendation in all iterations, precision values tended to get lower, especially when just a few relevant APIs are expected to be received in the recommendation lists because more irrelevant APIs can appear causing those low precision values, e.g., iteration 1 at $N = 20$ with a precision of 15.7%.

Regarding recall metric (Figure 82), we observed how these values increased as N increased. That result is normal and expected, since between more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, in all iterations the highest recall values were achieved. However, iteration 5 presented significant difference regarding the other iterations with recommendation lists for $N = 20$. Then, that

difference is normal since in software of iteration 5 there were from 12 to 73 relevant APIs and we cannot expect to receive all relevant APIs when just 20 were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 4.1% to 23.3%), again, because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 83), when we requested for small lists of APIs, e.g., $N = 1$ and $N = 3$, our recommendation methodology could correctly recommend at least one relevant API for more than 80% of request tests in all iterations. Besides, in lists from $N = 10$ to $N = 20$, our methodology recommend one relevant API to 100% of the requests for all iterations excepting iteration 2 where some irrelevant APIs were recommended, causing recall rate of 97.1%.

7.9 General discussion

When analyzing overall results in Stage A and in Stage B regarding large lists of APIs recommended, we expected maximum recall and low precision values based the mutual and inversely dependence between those metrics, since allowing longer recommendation lists typically improves recall and it is likely to reduce the precision (GUNAWARDANA; SHANI, 2009). On the other hand, we analyzed overall results of Stage A and Stage B considering small lists, i.e., varying *N* in top-*N* lists of APIs recommended. However, in very small *N* values, we decided to disregard recall metric, since for our recommendation methodology would be impossible recommend all relevant APIs when just few of them had been requested. Therefore, we discusses overall results of API recommendation for Stage A and Stage B as follows.

7.9.1 API recommendation for Stage A

As overall results for large lists in Stage A, we obtained promising results among the eight categories, e.g., in average, 56.7% of the relevant APIs were

recommended with precision of 31.3%. Besides, at least one relevant API for 99.8% of the requests was correctly recommended.

In small lists, e.g., in top-1 lists among the eight categories, we obtained averaged recall of 6.4%. Divergently, precision and recall rate metrics gave significant information in those small lists, e.g., the averaged precision in top-1 lists was 98.4%, meaning that just in few tests there were recommended irrelevant APIs. Besides, recall rate metric showed that our methodology could correctly recommend at least one relevant API for 98.7% of the requests among categories.

Regarding top-20 lists of APIs recommended, we did consider results of recall metric because, in some cases, the relevant APIs could fit in those lists. In all categories, we observed that recall values increased along with *N* and, in average, 47.1% of the relevant APIs were recommended in top-20. On the other hand, in all categories, we observed how precision values decreased as *N* increased and, in some cases, irrelevant APIs were recommended in top-20 lists causing an averaged precision of 45.6%. Besides, recall rate metric showed that our methodology could correctly recommend at least one relevant API for 99.8% of the requests among the eight categories.

On the other hand, considering the occurrence of every API over lists of APIs recommended in tests carried out on Stage A (Table 76) we observed that APIs rarely recommended (i.e., R rate) were apparently used for more specific features and services because they were recommended to few categories (Table 76). For example, API `javax.imageio.stream` is related to input/output of images and it was recommended to some software in Graphics category. Another example is `javax.mail`, an API related to build mail and messaging applications that was recommended to some software just in Communications category. In the same way, APIs partially recommended (i.e., P rate) provide specific features and services, but they are more common among categories. For example, API `java.sql` is used for accessing and processing data stored in databases and it

was recommended for some software in four categories. Moreover, in this ranked APIs, some of them are related to logging services, e.g., `org.apache.commons.logging`, `org.apache.log4j`, and `java.util.logging`. Finally, APIs rated as largely or fully recommended (i.e., L or F rate) are more common and well-known APIs and most of them are related to Graphical User Interface (GUI).

7.9.2 API recommendation for Stage B

As overall results for large lists in Stage B, we obtained promising results among the eight categories, e.g., in average, 52.6% of the relevant APIs were recommended with precision of 26.6%. Besides, at least one relevant API for 98.9% of the requests was correctly recommended.

In small lists, e.g., in top-1 lists, among the eight categories, we obtained averaged recall of 11.2%. Divergently, precision and recall rate metrics gave significant information in those small lists, e.g., the averaged precision in top-1 lists was 91.0%, meaning that just in few tests there were recommended irrelevant APIs. Besides, recall rate showed that our methodology could correctly recommend at least one relevant API for 91.0% of requests among the eight SourceForge categories.

Regarding top-20 lists of APIs recommended, we did consider results of recall metric because in some cases, the relevant APIs could fit in those lists, mainly because half of APIs were removed for simulating real users (Chapter 5). In all categories, we observed that recall values increased along with N and in average, 48.9% of relevant APIs were recommended in top-20 lists. On the other hand, in all categories, we observed how precision values decreased as N increased and, in some cases, irrelevant APIs were recommended in top-20 lists causing an averaged precision of 29.9%. Besides, recall rate metric showed that

our methodology could correctly recommend at least one relevant API for 98.3% of requests among the eight SourceForge categories.

On the other hand, considering the occurrence of every API over lists of APIs recommended in tests carried out on Stage B (Table 77) we observed that most APIs (i.e., 50.0%) were rarely recommended (i.e., R rate) and apparently, they are used for more specific features and services, because they were recommended to few categories (Table 77). For instance, APIs `org.lwjgl.opengl.glu` and `org.lwjgl.util.vector` were recommended to some software of Graphics category. Although those APIs are used for game development and part of LWJGL (Lightweight Java Game Library), they were recommended in Graphics category; apparently, because they handle with shapes, forms, objects, etc. and space positions as vectors matrices for objects (camera, image, player, etc.). Then, we assumed that they were recommended based on target software' similarity to model software from Graphics category.

8 THREATS TO VALIDITY

In this Chapter we discussed the main four threats to validity and the measures we deployed to minimize their impact. The remainder of this Chapter is organized as follows. Section 9.1 exposes construct validity. Section 9.2 discusses internal validity. Section 9.3 presents external validity. Section 9.4 shows conclusion validity.

8.1 External validity

It refers to the experiment ability to be generalized. In our study, the set of chosen software represents a threat to external validity. Firstly, we used just Java software developed in the IDE Eclipse. Nevertheless, we did not consider it as a threat, since Java is a programming language widely used in practice and the IDE Eclipse has garnered so much support to industry and it is now the key Java-tools (GEER, 2005). Besides, Eclipse includes rich Java Development Tools (JDT) support and a plug-in architecture that allows tight integration of third-party functionality (MURPHY; KERSTEN; FINDLATER, 2006).

Secondly, we expected to get a much larger scale of relevant software and conversely, from the ~6,632 non-empty software, only 830 achieved our data criteria. Besides, we could use just 315 from 830 software due to few numbers of software in Security & Utilities category (5 software) and due to large number of software in Development category (510 software) that overflowed not just our plug-in but also the Eclipse IDE. However, 315 is a reasonably fair set of software to conduct the study, which also has a larger scale compared to some related work (MILEVA et al., 2009; HERNÁNDEZ; COSTA, 2015; ASADUZZAMAN et al., 2015).

Thirdly, other threat is consequence of using JDT for developing the plug-in and Working Sets for categorizing software. However, we did not find any API

for accessing projects in other workspaces[10] and results showed that even with that limitation our methodology was able to make useful API recommendations.

Finally, the sample of software could not be the best representative, either for data criteria established or for categories. Nonetheless, we avoided subjective classification and we used the SourceForge categorization. Besides, we did a manual inspection in the SourceForge repository to every software, guaranteeing currently and correctness of category information;

8.2 Internal validity

It refers to experimenter prejudice. However, in our study, we automated our API recommendation methodology. We even simulated the users' behavior automatically and most of that process is randomized. Thus, we believed there is low experimenter bias.

8.3 Construct validity

It refers to relationship between theory proposed and results observe, i.e., in our study, it refers to the right eligibility method to we measure the effectiveness of our methodology. In our study, we used recall, precision, and recall rate evaluation metrics. Precision and recall metrics are the two best-known classification metrics (JANNACH et al., 2010). Those metrics measure to what extent a recommendation system is able to correctly classify items as interesting or not (ROBILLARD et al., 2014). Recall rate metric has a different nature, it responds with what percentage of cases the response was positive, i.e., it measures the efficacy (MALHEIROS, 2011). Besides, recall rate metric has been widely adopted by other studies and is the main evaluation metric to measure the efficacy

[10] https://stackoverflow.com/questions/29630407/how-can-i-set-the-iworkspace-root

of an API recommendation approach that is a baseline of our methodology (THUNG; LO; LAWALL, 2013).

8.4 Conclusion validity

Major of results were obtained through data given by the developed plug-in. Thus, implementation error could exist, but deep manually reviews were carried out and findings were coherent. Besides, before using real and relevant software, we developed many toy software projects for testing every module of the plug-in repetitively.

9 RELATED WORK

Many studies had used and demonstrated how frequency (popularity) helps Software Engineers in API issues. For example, for supporting Software Engineers with the decision of choosing the right API version, using a technique based on the popular vote of the majority where the more people use a particular version, the higher its usage is recommended, i.e., the most popular, more recommended. In their study, authors tried to recommend or dissuade from switching library versions based on global usage history (MILEVA et al., 2009). One of the disadvantages of this approach is the reasons that many Software Engineers had for using or switching from/to specific APIs in their development or maintenance. Hence, ignoring individual reasons could be a thread to validity when dissuading or recommending specific APIs. In contrast, our study recommends APIs based on their frequency of usage and based on the nearest (most similar) software considering software categories.

In another study, frequency of API use ("popularity") helped API Software Engineers to prioritize their bug-fixing efforts by repairing frequently-used APIs over "less important" APIs. It helps application Software Engineers to focus their investigative efforts on APIs that more developers have found useful in the past, rather than wade through large API descriptions to find what they need. In that work, popularity metric simply enumerates the number of times any particular API is used in in a specific way (HOLMES; WALKER, 2007). Similarly, in our study, we used the frequent itemset mining technique where the frequency of APIs was the number of times the API was used along the software categories or along the nearest (most similar) software.

On the other hand, software reuse is often achieved using frameworks and libraries, whose functionality is exported through APIs (ROBILLARD et al., 2013). However, effectively using APIs remains a challenge for Software

Engineers because they may not become aware of these APIs as they are released and developers may thus be led to “re-implement the wheel” (THUNG et al., 2013).

Because of that issue, studies have been made in order to support Software Engineers with APIs available. For example, to solve problems related to APIs in software maintenance, authors proposed a search engine of API experts called LIBTIC. That search engine served for the identification of experts in a given project of a library with the objective to ask them to perform tasks that require expertise. In addition, LIBTIC served to identify if there were experts that can be contacted to answer some very specific questions about APIs. In the tested scenarios, LIBTIC was able to return relevant developer profiles, reaching the needs of the search for API experts (TEYTON et al., 2013a). Although it is helpful for dealing with API troubles, the solution is partial, because identifying APIs experts requires investment, availability, and human resources that may not be included within the budget of software development or maintenance.

However, other studies and approaches avoid that dependency of external human resources, availability, or extra investment by supporting Software Engineers through recommendation techniques. The first example is Precise, an automated approach to parameter recommendation for API usage, which is able to recommend API parameters frequently used in practice. Therefore, during programming, when the developer has already selected a method and is going to determine the actual parameter(s) of a method, Precise uses the algorithm of k-Nearest Neighbors to find instances of parameters commonly used in similar contexts in the pasts, and automatically recommends a list of well-sorted parameter candidates (i.e., by frequency) (ZHANG et al., 2012).

The second example is a hybrid approach that combines association rule mining and collaborative filtering techniques to assist Software Engineers in the correct choice of APIs. Thus, APIs were recommended to Software Engineers

based on the API usage patterns and based on those that are used by other similar software. That approach presented promising results since the recall rate@5 and recall rate@10 were 0.852 and 0.894, respectively (THUNG; LO; LAWALL, 2013).

Although, a key constraint of this strategy is the fact of exclusively recommending APIs for software that used some APIs, disregarding those that did not. One strategy for addressing that main constraint is considering other factors like API migrations, released time, specific reasons of API usage, categories of APIs, and others (TEYTON et al., 2013b).

In a similar way to those studies exposed above, we used a methodology for recommending APIs to developers and Software Engineers by using Frequent Itemset Mining and Collaborative Filtering techniques. In our study, we considered SourceForge categories as Games, Audio & Video, Graphics, and others. Besides, we considered software in initial stage of development (APIs are not used) and in advanced stage of development (some APIs are used). On the other hand, we used not just the recall rate metric to measure how promising our results were; also, we used precision and recall metrics to complement our evaluation of the recommendation system. As consequence, we demonstrated that our methodology was able to recommend useful APIs to Software Engineers of Java software in the IDE Eclipse.

10 FINAL REMARKS

Recommendation systems support users on making-decisions. In the Software Engineering area, they can support development and maintenance activities, providing benefits like increasing Software Engineers' productivity and software quality, and minimizing their effort in tasks related to software components, code, human resources, and others. An essential component for software development is the Application Programming Interface (API), that lets to reuse features and services letting to improve Software Engineers' productivity since they do not need to develop features already available for using. However, choosing the right APIs is not an easy task.

The remainder of this Chapter is organized as follows. Section 11.1 concludes our study. Section 11.2 presents contributions. Section 11.3 proposes future work.

10.1 Conclusion

We defined a methodology that considered categories and automatically recommends APIs to Software Engineers in initial stage (Stage A, i.e., still do not use APIs at all) or in advanced stage (Stage B, i.e., use some APIs already) of software development. We used Collaborative Filtering technique (CF) along with Frequent Itemset Mining technique (FIS). Nevertheless, in Stage A, CF was not an appropriate choice because target software of recommendations did not use any API; so, it was not possible to relate them with the model software. Thus, in Stage A, we chose FIS mining technique to overcome that limitation. On the other hand, in Stage B, we used both techniques.

In order to automate that methodology, we developed a plug-in for the IDE Eclipse and we considered application categories from the SourceForge open source repository. We did not involve real user's interaction for obtaining

"immediate" results. Then, we evaluated the quality of our recommendation methodology using system-centric evaluation. In addition, we used a 5-fold cross validation in every category and two classification accuracy metrics (precision and recall) and one efficacy metric (recall rate) for evaluating the recommendation task for large lists and top-*N* lists of APIs recommended. In top-*N* lists, the goal was to find a few specific APIs supposed to be most useful.

As results, our evaluation showed that our methodology could carry out useful API recommendations for Software Engineers with software that use a small number of APIs or do not use any APIs, even in small top-*N* lists of APIs recommended. In addition, results presented promising outcomes for each category. In Stage A, regarding large lists, averaged metric values were 56.7% of recall, 31.3% of precision and 99.8% of recall rate. Regarding top-20 lists, averaged metric values were 47.1% of recall, 45.6% of precision and 99.8% of recall rate. On the other hand, in Stage B, regarding large lists, averaged metric values were 52.6% of recall, 26.6% of precision and 98.9% of recall rate. Regarding top-20 lists, averaged metric values were 48.9% of recall, 29.9% of precision and 98.3% of recall rate.

10.2 Contribution

We contributed to software engineering by proposing an API recommendation methodology that partially overcame the cold-star problem, i.e., recommending useful APIs to Software Engineers with software that did not even use any API. As benefits, we expected to support Software Engineers in decision-making process about the right APIs to use in their software development and/or maintenance. We also contributed by providing a plug-in for the IDE Eclipse that implements that API recommendation methodology.

Although we did not evaluate some other possible benefits like the increment in Software Engineers' productivity and in software quality, we expect

them to be consequences of using our recommendation system, since features and services provided by the APIs recommended will be reused. Thus, development time would be minimized and popularity of APIs could be an indicator of their quality and thus, quality of software could be increased.

10.3 Future work

In future, we plan to make a deeper qualitative analysis for Stage A and Stage B, discussing frequency of APIs recommended in every category and discussing results found from applying the 4-point rating scale to APIs recommended among the eight categories. In addition, we intend to discuss results between Stage A and Stage B for every category, analyzing differences and improvements provided by each strategy used.

On the other hand, as part of the data collection process in our API recommendation methodology (Section 5.2), we considered all import statements by removing their package members (classes or interfaces) statements because a Java API is a collection of packages. We also differentiated own software packages from external APIs packages in the import statements. However, we suggest processing APIs statements before making the recommendations in order to recommend the entire package without their simplified name, i.e., recommending general APIs.

For API recommendation for Stage A (Chapter 8), we presented our strategy to select the minimum values of support (i.e., *minSupport* value). We aimed to select the correct value for every category, while maximizing the coverage of APIs recommended as possible. As a result, we obtained reasonable number of APIs recommended, excepting Business & Enterprise category, due to the high *minSupport* value selected. Then, as future work, we suggest to use different strategies for selecting the *minSupport* value to compare its influence in correctness and coverage of the API recommendation methodology.

In addition, we used a strategy for simulating users' behavior where we decided to remove 50% of APIs from every target system, saving them as the relevant APIs expected to be recommended by our methodology (Section 5.4). Then, as future work, we suggest to vary that percentage of APIs removed among recommendation tests in order to compare its influence on coverage and quality of APIs recommended.

Furthermore, we used the well-known similarity metric Jaccard for computing Nearest Neighbors software (Section 5.3). Then, other suggestion is using other known similarity metrics, e.g., Cosine and Pearson. Thus, a comparison would be possible in order to measure correctness, optimization, and quality of APIs recommended.

We used the JDT for developing the plug-in with the recommendation methodology and the Working Sets for categorizing software. That strategy generated threats to validity (Section 9.1). Consequently, to overcome those threats we suggest to develop a standalone API recommendation system.

As future work, we suggest to perform a controlled experiment with real users (API experts and inexpert) to check if their API reuse decision is influenced by the results of our recommendations. At the same time, as real users would be involved, a study to measure their productivity before and after using the recommendation system could be made.

On the other hand, we considered application categories as a main requirement of our recommendation methodology, and as future work, domains could be used. Therefore, instead or along with SourceForge categories, API domains could be considered, e.g., logging, testing, persistence, etc. Besides, in order to find more relevant software, same methodology could be used with different data sample from other well-known open source repositories like GitHub, Google code, CodePlex, etc.

REFERENCES

ACHARYA, M. et al. Mining API patterns as partial orders from source code : from usage scenarios to specifications. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 6., 2007, Cavat. **Proceedings**... New York: ACM, 2007. p. 25-34.

AGRAWAL, R.; IMIELINSKI, T.; SWAMI, A. Mining association rules between sets of items in large databases. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 93., 1993, New York. **Proceedings**... New York: ACM, 1993. p. 207-216.

ALHARTHI, H. **The use of items personality profiles in recommender systems**. 2015. Cap. 5. Dissertation (Master in Computer Science) - University of Ottawa. Faculty of Graduate and Postdoctoral Studies, Ottawa, 2015.

ANAND, D.; BHARADWAJ, K. K. Utilizing various sparsity measures for enhancing accuracy of collaborative recommender systems based on local and global similarities. **Expert Systems With Applications**: an international journal, Tarrytown, v. 38, n. 5, p. 5101-5109, May 2011.

ASADUZZAMAN, M. et al. Exploring API method parameter recommendations. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, 2015, Bremen. **Proceedings...** Bremen: IEEE, 2015. p. 271-280.

BAJRACHARYA, S.; OSSHER, J.; LOPES, C. Sourcerer: an infrastructure for large-scale collection and analysis of open-source code. **Science of Computer Programming**, Amsterdam, p. 241-259, Jan. 2014. Available in: <https://www.info.fundp.ac.be/wasdett2010/wp-content/uploads/2010/08/BOL-WASDeTT3.pdf>. Access in: 10 dec. 2015.

BAJRACHARYA, S.; OSSHER, J.; LOPES, C. Sourcerer: an internet-scale software repository. In: SEARCH-DRIVEN DEVELOPMENT-USERS, INFRASTRUCTURE, TOOLS AND EVALUATION, 9., 2009, Vancouver. **Proceedings**... Vancouver: IEEE, 2009. p. 1-4.

BIGDELI, E.; BAHMANI, Z. Comparing accuracy of cosine-based similarity and correlation-based similarity algorithms in tourism recommender systems. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF INNOVATION AND TECHNOLOGY, 4., 2008, Bangkok. **Proceedings**... Bangkok: IEEE, 2008. p. 469-474.

BOBADILLA, J. et al. A framework for collaborative filtering recommender systems. **Expert Systems With Applications**: an international journal, Tarrytown, v. 38, n. 12, p. 14.609-14.623, Nov. 2011.

BOBADILLA, J. et al. Recommender systems survey. **Knowledge-Based Systems**, Amsterdam, v. 46, p. 109-132, July 2013.

BREESE, J. S.; HECKERMAN, D.; KADIE, C. Empirical analysis of predictive algorithms for collaborative filtering. In: CONFERENCE ON UNCERTAINTY IN ARTIFICIAL INTELLIGENCE, 14., 1998, San Francisco. **Proceedings**... San Francisco: Morgan Kaufmann, 1998. p. 43-52.

CANÓS, J. H.; LETELIER, P.; PENADÉS, M. C. Metodologías ágiles en el desarrollo de software. In: TORRES, P. L.; LÓPEZ, E. A. S. (Ed.). **Metodologías ágiles en el desarrollo de software**. Valencia: ISSI/JISBD, 2003. p. 1-8. (Actas). Available in: <http://issi.dsic.upv.es/archives/f-1069167248521/actas.pdf>. Access in: 10 dec. 2015.

CREMONESI, P. et al. An evaluation methodology for collaborative recommender systems. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOLUTIONS FOR CROSS MEDIA CONTENT AND MULTI-CHANNEL DISTRIBUTION, 4., 2008, Florence. **Proceedings**... Florence: IEEE, 2008. p. 224-231.

CREMONESI, P.; GARZOTTO, F.; TURRIN, R. User-Centric vs. system-centric evaluation of recommender systems. In: KOTZÉ, P. et al. (Ed.). **Human-Computer Interaction** – INTERACT 2013. Milano: Springer Berlin Heidelberg, 2013. Part 3, p. 334-351. (Lecture Notes in Computer Science, 8119-2013).

DAPENG, H.; QIANHUI, L.; JINGMIN, Z. An improved similarity algorithm for personalized recommendation. In: INTERNATIONAL FORUM ON

COMPUTER SCIENCE-TECHNOLOGY AND APPLICATIONS, 9., 2009, Chongqing. **Proceedings...** Chongqing: IEEE, 2009. p. 54-57.

DUALA-EKOKO, E.; ROBILLARD, M. P. Asking and answering questions about unfamiliar APIs: an exploratory study. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34., 2012, Zurich. **Proceedings**... Piscataway: IEEE, 2012. p. 266-276.

GALÁN, S. M. Filtrado colaborativo y sistemas de recomendación. In: INTELIGENCIA EN REDES DE COMUNICACIONES, 2007, Légames. **Proceedings**... Légames: Universidad Carlos III de Madrid, 2007. p. 261-268.

GEER, D. Eclipse becomes the dominant Java IDE. **Computer**, Washington, v. 38, n. 7, p. 16-18, July 2005.

GOLDBERG, D. et al. Using collaborative filtering to weave an information tapestry. **Communications Of The ACM**, New York, v. 35, n. 12, p. 61-70, Dec. 1992.

GUNAWARDANA, A.; SHANI, G. A survey of accuracy evaluation metrics of recommendation tasks. **Journal of Machine Learning Research**, v. 10, p. 2935-2962, Dec. 2009. Available in: <https://www.ischool.utexas.edu/~i385d/readings/Goldberg_UsingCollaborative_92.pdf >. Access in: 10 dec. 2015.

GUO-RONG, L.; XI-ZHENG, Z. Collaborative filtering based recommendation system for product bundling. In: INTERNATIONAL CONFERENCE ON MANAGEMENT SCIENCE AND ENGINEERING, 2006, Lille. **Proceedings**... Lille: IEEE, 2006. p. 251-254.

HERNÁNDEZ, F.; GAUDIOSO, E. Evaluation of recommender systems: a new approach. **Expert Systems With Applications**: an international journal, Tarrytown, v. 35, n. 3, p. 790-804, Oct. 2008.

HERNÁNDEZ, L.; COSTA, H. Identifying similarity of software in apache ecosystem - an exploratory study. In: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY - NEW GENERATIONS, 12., 2015, Las

Vegas. **Proceedings**... Las Vegas: IEEE, 2015. p. 397-402.

HOLMES, R.; WALKER, R. J. Informing eclipse API production and consumption. In: OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 2007, Montréal. **Proceedings**... New York: ACM, 2007. p. 70-74.

HUANG, A. Similarity measures for text document clustering. In: NEW ZEALAND COMPUTER SCIENCE RESEARCH STUDENT CONFERENCE, 6., 2008, Christchurch. **Proceedings**... Christchurch: NZCSRSC, 2008. p. 49-56.

JANNACH, D. et al. Evaluating recommender systems. In: JANNACH, D. et al. **Recommender systems**: an introduction. New York: Cambridge University, 2010. Cap. 7. p. 166-188.

JANNACH, D.; ZANKER, M. **Tutorial**: evaluation of recommender systems - methodology. 2012. At ACM Symposium on Applied Computing - SAC. Available in: <https://pdfs.semanticscholar.org/f33c/507310f9a16b2ed0efcbf983804e610952af.pdf>. Access in: 1 mar. 2015.

KARYPIS, G. Evaluation of item-based Top-N recommendation algorithms. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, 10., 2001, Mclean. **Proceedings**... New York: ACM, 2001. p. 247-254.

LIU, H. et al. A new user similarity model to improve the accuracy of collaborative filtering. **Knowledge-Based Systems**, Amsterdam, v. 56, p. 156-166, Jan. 2014.

LOPES, C. et al. **UCI Source Code Data Sets**: SDS_source-repo-18k. 2010. Irvine, CA: University of California/Bren School of Information and Computer Sciences. Available in: <http://www.ics.uci.edu/~lopes/datasets>. Access in: 15 jan. 2015.

MAFFORT, C. et al. Mining architectural patterns using association rules. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 25., 2013, Boston. **Proceedings**... Boston: SEKE, 2013. p. 375-380.

MALHEIROS, Y. de A. **Um sistema de recomendação de código-fonte para suporte a novatos**. 2011. Cap. 4, 87 f. Dissertation (Master Ciência da Computação) - Universidade Federal de Pernambuco. Centro de Informática, Recife, 2011.

MARTINS, R. J. W. de A. **Recomendação de pessoas em redes sociais com base em conexões entre usuários**. 2013. Cap. 2, 59 f. Dissertation (Master Informática) - Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática do Centro Técnico Científico, Rio de Janeiro, 2013.

MILEVA, Y. M. et al. Mining trends of library usage. In: JOINT INTERNATIONAL AND ANNUAL ERCIM WORKSHOPS ON PRINCIPLES OF SOFTWARE EVOLUTION (IWPSE) AND SOFTWARE EVOLUTION (EVOL) WORKSHOPS, 2009, Amsterdam. **Proceedings**... New York: ACM, 2009. p. 57-62.

MONTANDON, J. E. et al. Documenting APIs with examples: lessons learned with the APIMiner platform. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 20., 2013, Koblenz. **Proceedings**... Koblenz: IEEE, 2013. p. 401-408.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are Java software developers using the Eclipse IDE? **IEEE Software**, Los Alamitos, v. 23, n. 4, p. 76-83, July 2006.

NIWATTANAKUL, S. et al. Using of jaccard coefficient for keywords similarity. In: THE INTERNATIONAL MULTICONFERENCE OF ENGINEERS AND COMPUTER SCIENTISTS, 2013, Hong Kong. **Proceedings**... Hong Kong: IAENG, 2013. p. 380-384.

NÚNEZ-VALDÉZ, E. R. et al. Plataforma de recomendación de contenidos para libros electrónicos inteligentes basada en el comportamiento de los usuarios.**Technology Journal LAC**: the next trend of technology, Lima, p. 25-40, Mar. 2012. Available in: <http://www.udla.edu.co/revistas/autores/Comite%20editorial/Carlos%20Enrique%20Montenegro%20Mar%C3%ADn/2012(2).pdf>. Access in: 10 dec. 2015.

PARAMBATH, S. A. **Matrix factorization methods for recommender**

systems. 2013. Cap. 2, 36 f. Dissertation (Master in Computer Science) - Umeå University, Sweden, 2013.

PARMAR, A.; SUTARIA, K.; JOSHI, K. An approach for finding frequent item set done by comparison based technique. **International Journal Of Computer Science And Mobile Computing**, v. 3, n. 4, p. 996-1001, Apr. 2014. Available in: <http://www.ijcsmc.com/docs/papers/April2014/V3I4201499b37.pdf>. Access in: 10 dec. 2015.

RICCI, F. et al. (Ed.). **Recommender systems handbook**. Boston: Springer, 2011. Cap. 1. p. 10-14.

ROBILLARD, M. P. et al. Automated API property inference techniques. **IEEE Transactions on Software Engineering**, Piscataway, p. 613-637, May 2013.

ROBILLARD, M. P.; WALKER, R.; ZIMMERMANN, T. Recommendation systems for software engineering. **IEEE Software**, Washington, v. 27, n. 4, p. 80-86, June 2010.

ROBILLARD, M. P. et al. **Recommendation systems in software engineering**. Berlin: Springer, 2014. 561 p.

RUPAKHETI, C. R.; HOU, D. Satisfying programmers' information needs in API-based programming. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 19., 2011, Kingston. **Proceedings...** Kingston: IEEE, 2011. p. 250 - 253.

SAEED, M. et al. Software clustering techniques and the use of combined algorithm. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 7., 2003, Benevento. **Proceedings...** Washington: IEEE, 2003. p. 301-306.

SCHRÖDER, G.; THIELE, M.; LEHNER, W. Setting goals and choosing metrics for recommender system evaluations. In: UCERSTI, 2; WORKSHOP AT THE 5; ACM CONFERENCE ON RECOMMENDER SYSTEMS, 5., 2011, Chicago. **Proceedings**... Chicago: UCERSTI, 2011. p. 78-85. Available in: <http://ucersti.ieis.tue.nl/files/papers/4.pdf>. Access in: 15 dec. 2015.

SON, L. H. Dealing with the new user cold-start problem in recommender systems: a comparative review. **Information Systems**: databases: their creation, management and utilization, p. 1-18, Dec. 2014.

SOUZA, B. de F. M. e. **Modelos de fatoração matricial para recomendação de vídeos**. 2011. Cap. 2, 59 f. Dissertation (Master Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2011.

STYLOS, J.; MYERS, B. Mapping the space of API design decisions. In: SYMPOSIUM ON VISUAL LANGUAGES AND HUMAN-CENTRIC COMPUTING, 2007, Coeur D'alene. **Proceedings**... Washington: IEEE, 2007. p. 50-60.

SUN, C.; KHOO, S.-C.; ZHANG, S. J. Graph-based detection of library API imitations. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 27., 2011, Williamsburg. **Proceeding**s... Williamsburg: IEEE, 2011. p. 183-192.

TEWARI, A. S.; KUMAR, A.; BARMAN, A. G. Book recommendation system based on combine features of content based filtering, collaborative filtering and association rule mining. In: INTERNATIONAL ADVANCE COMPUTING CONFERENCE, 2014, Gurgaon. **Proceedings**... Gurgaon: IEEE, 2014. p. 500-503.

TEYTON, C. et al. A study of library migration in Java Software. **CoRR**, Ithaca, p. 1-20, June 2013b. Available in: <http://arxiv.org/pdf/1306.6262.pdf>. Access in: 20 aug. 2014

TEYTON, C. et al. Find your library experts. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 20., 2013, Koblenz. **Proceedings**... Koblenz: IEEE, 2013a. p. 202-211.

TEYTON, C.; FALLERI, J.-R.; BLANC, X. Automatic discovery of function mappings between similar libraries. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 20., 2013, Koblenz. **Proceedings**... Koblenz: IEEE, 2013. p. 192 - 201.

THE JAVA TUTORIALS: ORACLE. **Using package members**. Available in:

<https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html>. Access in: 20 jun. 2014.

THUNG, F.; LO, D.; LAWALL, J. Automated library recommendation. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 20., 2013, Koblenz. **Proceedings**... Koblenz: IEEE, 2013. p. 182-191.

THUNG, F. et al. Automatic recommendation of API methods from feature requests. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 28., 2013, Silicon Valley. **Proceedings**... Silicon Valle: IEEE, 2013. p. 290-300.

ZHANG, C. et al. Automatic parameter recommendation for practical API usage. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34., 2012, Zurich. **Proceedings**... Piscataway: IEEE, 2012. p. 826-836.

ZUVA, T. et al. Survey of recommender systems techniques, challenges and evaluation metrics. **International Journal of Emerging Technology and Advanced Engineering**, v. 2, n. 11, p. 382-386, Nov. 2012. Available in: <http://www.ijetae.com/files/Volume2Issue11/IJETAE_1112_59.pdf>. Access in: 20 jun. 2014.