# API Recommendation System for Software - Game Category

**Luisa Hernández, Paulo Afonso Júnior, Heitor Costa**

Departamento de Ciência da Computação - Universidade Federal de Lavras - MG - Brazil

lufe.hernandez@gmail.com, pauloa.junior@dcc.ufla.br, heitor@dcc.ufla.br

***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. Considering that recommendation systems are emerging to support software engineers in their decision-making task and Games industry has a huge economic and cultural success, we proposed a technique that considers Game category from SourceForge 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. We evaluated 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 technique could make useful API recommendations for software engineers with Game software that used a small number of APIs or did not use any API. Besides, our technique was able to put relevant APIs even in high-ranking positions, even in small top-N lists of APIs recommended.*

## 1. Introduction

Software development is inseparable from the use of Application Programing Interfaces (APIs) [Duala-Ekoko; Robillard, 2012] due to the advantages of reusing them. For instance, APIs provide cost effective way to build software with enhance in [Sun *et al.*, 2011]: i) productivity of programmers by providing variety of desired functions; and ii) software quality, as libraries are usually well-tested and fairly robust because of their massive and diverse user base. Nevertheless, the increasing size and number of APIs implies the developers must frequently learn how to use the unfamiliar APIs [Duala-Ekoko; Robillard, 2012]. Hence, to reuse effectively and correctly APIs during the development is difficult for software developers [Acharya *et al.*, 2007]. An alternative is to find API experts [Teyton *et al.* 2013], but hiring those experts implies human and financial high investment that could not be included in development planning. Several studies have addressed the issue of APIs usage in order to guide and support developers [Mileva *et al.*, 2009; Duala-Ekoko; Robillard, 2012; Thung *et al.*, 2013]. However, to choose right APIs remains as a difficult task for developers and Recommendation Systems (RSs) emerged for solving those tasks.

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 developers in various activities [Robillard *et al.*, 2010]. For example, decision-making about which software components (or APIs) to reuse. Among advantages of using RSs and RSSEs, we cited reduction in effort, increment of productivity in software engineering activities, and supporting in users decision-making [Robillard *et al.*, 2014]. Moreover, they allow discovering new content faster and more efficiently [Núnez-valdéz *et al.*, 2012].

A study base for this work proposed a hybrid approach to recommend APIs, combining association rule mining and nearest neighbor based on collaborative filtering recommendation technique [Thung *et al.*, 2013]. In spite of the promising results presented in that study, we observed some limitations that motivated this current study. On the other hand, there are several repositories of software systems of open source (SourceForge, Google Code, Apache Software Foundation, and GitHub), which cluster those systems by application categories, programming languages, language, and/or user evaluation. We proposed a technique and we conducted empirical evaluations considering SourceForge software from Game category.

Therefore, the main motivation and differential of our recommender technique regarding other several studies found lies in considering: i) Java software (even Maven) developed in Eclipse-IDE; ii) Game category from SourceForge; iii) software in initial stage of development (in designing or with code but without use of APIs); and iv) software in advanced stage of development (with code and using few APIs). In addition, we simulated software engineers' scenarios for initial and advanced stage of software development. Results using our technique demonstrated that it was possible to obtain useful large and top-*N* lists of APIs recommended for software engineers whose systems belonged to Game category and used a small number of APIs or did not use any APIs.

The remainder of this paper is organized as follows. Section 2 summarizes background concepts. Section 3 presents technique to recommend APIs. Section 4 presents evaluation strategy. Section 5 shows the results. Section 6 discusses results. Section 7 presents some related work. Section 8 concludes and presents future work.

## 2. Background

### 2.1 Collaborative Filtering

Collaborative Filtering (CF) is the most widely used and effective recommendation technique and it is based on the assumption that users who have agreed in the past tend to agree in the future [Anand; Bharadwaj, 2011]. CF has been used in many real systems, e.g. environmental sensing, financial services, and web applications [Thung *et al.*, 2013].

One of the most frequently cited algorithm and possibly the most widely implemented is the Nearest Neighbor (NN) algorithm [McLaughlin; Herlocker, 2004]. Assuming that an entity is an item or a user, a basic method to perform CF technique finds similarities among entities and selects the most similar entities as the NN of the target entity of recommendation. The key for NN algorithms is the similarity calculation [Dapeng *et al.*, 2009]. Traditional similarity metrics (e.g., Cosine, Jaccard, and Pearson Correlation) can be used for computing that similarity [Dapeng *et al.*, 2009; Bodadilla *et al.*, 2011]. In this study, we used Jaccard, where similarity between two data sets of entities is the result of division between the number of common features and the number of properties [Niwattanakul *et al.*, 2013]. Jaccard ranges between 0 and 1, where 0 means dissimilarity and 1 means total similarity. After computing similarity, we can find the top-*N* highly ranked entities and return them as the top-*N* of entities recommended. In our context, an entity is one software and similarity is computed based on their APIs.

### 2.2 Frequent Item Set Mining

Frequent item set mining (FIS) is a data mining technique, which efficiently finds frequent item sets in a dataset. FIS defines the support as the number of occurrences of a subset of items (sub-item set). A sub-item set is frequent if its support is greater than a specified threshold called minimum support. Thus, the support is the number of times a sub-item set happens in the item sets database [Maffort *et al.*, 2013]. In this study, we computed the support of an API by counting the times that it is used in Game software.

### 2.3 Evaluation Metrics

The quality of Recommendation Systems (RSs) can be defined either in terms of system-centric method, which are evaluated algorithmically (e.g., with classification accuracy metrics - precision and recall) or with user-centric experiments (e.g., users interact with RS and receive recommendations) [Cremonesi *et al.*, 2013]. In order to have immediate results, in an independent and economical way, we decided to use system-centric method for evaluating recommendation technique, i.e., avoiding dependence of real users' interaction. Consequently, we used an efficacy metric [Barbosa, 2011] and two classification accuracy metrics for measuring to what extent a recommendation system was able correctly to classify items as interesting or not [Robillard *et al.*, 2014].

The purpose of a classification task in the context of item recommendation is to identify the top-*N* most relevant items for users. Two best-known classification metrics are Precision, probability that a recommended item corresponds to the user's preferences, and Recall, probability that a relevant item is recommended [Jannach *et al.*, 2010]. These metrics are suitable for evaluating top-*N* lists. When a recommender algorithm predicts the top-*N* items that a user expected to find interesting by using recall, we can compute the percentage of known relevant items from the test set that appear in the *N* predicted items [Cremonesi *et al.*, 2008]. On the other hand, recall rate@*k* is another useful metric for evaluating effectiveness of recommendation systems. Recall rate@*k* is proportion of top-*k* recommended lists, in the set of all recommendations (for all projects) that includes at least one relevant item recommended [Thung *et al.*, 2013]. This metric has a different nature, it responds with what percentage of cases the answer was positive, i.e., measures the efficacy [Barbosa, 2011]. Then, for computing recall rate@*k*, where *k* is the number of the recommended items, we must assign value 1 if, at least, one of the *k* recommended items (e.g. API) is a member of relevant items or value 0, otherwise [Thung *et al.*, 2013].

## 3. API Recommendation for Software from Game Category

No reference architecture has emerged to-date for RSSEs [Robillard *et al.*, 2014]. Nevertheless, there are three major processes in the RSs [Bigdeli; Bahmani, 2008]: i) **object data collections and representations** - consist in getting the items for analyzing; ii) **similarity decisions** - involve calculation of distance/similarity among data; and iii) **recommendation computations** - introduce and filter recommendation of relevant items. Thus, in a similar way, we established three main steps into a technique in order to support software engineers through large and top-*N* lists of APIs recommended for developing/maintaining software from Game category: i) Software Dataset; ii) Data Collection; and iii) Recommendation Engine.

## 3.1 Software Dataset

We used Sourcerer repository in this study, which is part of the source code repository UCI [Bajracharya *et al.*, 2009; Lopes *et al.*, 2010]. It contains ~18,826 (~13,241 non-empty) software from Apache, Java.net, Google Code, and SourceForge repositories. We decided to use SourceForge dataset, which has 9,969 (~6,632 non-empty) software among 10 categories provided. Therefore, we determined other criteria to get relevant Game software. In this context, we determined as relevant software that: i) were developed in Java and Eclipse IDE platform; ii) used at least two APIs; iii) were independent (not embedded software); iv) had been registered for at least four years; v) whose last date of maintenance between 2010 and 2015; and vi) belonged exclusively from Game category. After filtering, we found 70 software and we stored them in a main repository.

## 3.2 Data Collection

It consists in getting the items for analyzing them. In our study, items are APIs from target software of recommendation and model software, both cases regarding software categories. Within Java software, there are different ways to capture API information, for example, by: i) corresponding binary files (.jar files); ii) importing statement in each .java file where the corresponding class term should be disregarded; and iii) declared tags of <dependency> via pom.xml files in Maven systems.

We decided to capture APIs through importing statement in the .java files. That statement allows reusing existing features, besides there are two ways to reference them [The Java Tutorials, 2014]: i) entire package using the '*' character referencing all of the members (class or interface) contained in the package (e.g. import graphics.*); or ii) package members (class or interface) using their simplified name (e.g. import graphics.Circle). In addition, we determined to remove the statements corresponding to the package members (classes or interfaces), whereas a Java API is a collection of packages. Other main decision was to differentiate own system packages in the import statements and the external APIs packages in the import statements.

## 3.3 Recommendation Engine

We created two stages for the recommendation engine because we must consider the cases that were not possible to establish similarities. Therefore, the recommendation engine considers Game software and consists on strategies and techniques used to obtain the relevant APIs considering two stages of software development: i) **Stage A**: API Recommendation for software engineers in initial stage of software development (not using APIs); and ii) **Stage B**: API Recommendation for software engineers in advanced stage of software development (using some APIs).

For **Stage A**, recommendation techniques were not a good choice because there were no APIs to link between Game model software and the Game target software of API recommendation. In recommender systems that issue is known as cold-start problem [Schein *et al.*, 2002; Park *et al.*, 2006; Son, 2014]. However, in order to recommend APIs to developers with systems in initial stage of development, we decided to consider the popularity of APIs in Game category. One way to determine that popularity was using FIS technique for establishing the most common APIs in Game software and relating results to the target software of API recommendation. Once that popularity of APIs is computed, the final large list of APIs recommended can be generated along with the top-

*N* lists. In both lists, APIs are sorted by its popularity value. Regarding **Stage B**, we used NN algorithm from CF in conjunction with FIS technique, i.e., considering model software from Game category. Thereafter, the API similarity is computed between the Game software and the target software of API recommendation. As a result, the nearest systems are found and the list of possible APIs to recommend can be generated. Afterward, FIS technique may be applied and union can be made between the lists from CF and FIS techniques. The final lists of APIs recommended can be generated along with the top-*N* list, which is sorted by every API popularity value.

## 4. Evaluation Strategy

We determined to evaluate our technique through system-centric evaluation, without real user's interaction, because conducting empirical tests involving real users is difficult, expensive, and resource demanding. On the contrary, system-centric evaluation has the advantage to be immediate, economical and easy to perform on several domains and with multiple algorithms [Cremonesi *et al.*, 2013]. We simulated software engineer scenarios for Stage A and Stage B of software development. Then, based on each stage, we did dataset partition of 5-fold cross validation [Cremonesi *et al.*, 2008], i.e., we split dataset into 80% Training (Game model software) and 20% Test data (software to simulate users). Hereafter, we will call them as *M-software* and *TS-software* respectively.

### 4.1 Evaluation of Stage A of API Recommendation

We defined to remove the 100% from the *TS-software* in order to simulate software engineers in initial stage of software development or maintenance. We saved the removed APIs in .xml files and saved them as relevant APIs (APIs that we expect to recommend). In addition, we varied attributes the threshold (*minSupport*) value for FIS technique from 0.1 to 0.9 and we found the best threshold value to use for generating corresponding large and top-*N* lists of APIs recommended. Afterwards, for evaluating the quality of our API recommendation technique, we applied recall, precision, and recall rate (evaluation metrics) based on the .xml files of APIs. Then, two recommendation lists generated (one list with large recommendations and the other list with the top-*N* recommendations). In case of Top-*N* lists, we varied *N* from 1 to 20 in order to study how early correct recommendations appear in the top-N recommendation. Metric values are generated in .xls files for further manual analysis.

### 4.2 Evaluation of Stage B of API Recommendation

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 relevant APIs (APIs that we expect to recommend). As that removal is made randomly, we established to do five replicates of API recommendation for every software of the test set in every iteration. We used CF technique along with FIS technique; we needed to find the "best" value for each attribute. In case of *minSupport*, we varied *minSupport* value for FIS technique from 0.1 to 0.9 and we found the best threshold value to use for generating corresponding lists of APIs recommended by FIS technique. In case of CF technique, we varied the *k* number of nearest neighbors (*kNN*) in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method in order to find the best *kNN* for generating the lists of APIs recommended lists by CF technique. Subsequently, the union between FIS and CF lists of APIs recommended is made, regarding that each list was already sorted by *support* value. For that union, we assigned an equitable weight of 50% of relevance for every strategy. Two lists of APIs recommended are generated, one with the union of all API recommendations and the other with the top-*N* list of recommendation.

## 5. Results

In our empirical evaluation, we used a dataset of 70 software exclusive from Game category. In Table 1, we showed software and their number of APIs. In order to evaluate our recommendation technique for software engineers with software categorized in Game category regardless of the stage of software development, we did the 5-fold cross validation. Hence, in every iteration, we used 14 different software (20% of data) as test set and 56 software (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. Hence, we applied the evaluation strategy for Stage A and Stage B separately. Then, we computed averaged recall, precision, and recall rate values in order to analyze the quality of our recommendation technique in every stage of API recommendations.

**Table 1. Baseline Data for Game Category**

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

### 5.1. API Recommendation for Game Software in Stage A

In this Stage, in each iteration of the 5-fold cross validation, for each software from test set, all of APIs were removed and saved in one .xml file as relevant APIs (APIs that we expect to recommend). The core is the FIS technique; thus, we needed to find the "best" threshold (*minSupport*) value. So, we analyzed the effect of varying this value of FIS technique (from 0.1 to 0.9) in top-10 lists of APIs recommended (Table 2). 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 since choosing other value did not mean any significant improvement for precision or recall rate values.

After finding *minSupport* value, we used it for evaluating our technique for recommending APIs to software engineers whose software is from Game 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 2. Effect of Varying *minSupport* Value in Top-10 Lists of APIs Recommended in Stage A of Game 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 |

Regarding the evaluation when receiving large list of APIs recommended, we obtained the averaged evaluation results (recall, precision, and recall rate) from iterations of the 5-fold cross validation (Recall = 0.660; Precision = 0.337; Recall Rate = 1.000). In addition, for analyzing every iteration, we exposed the evaluation results in Figure 2. 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 3). In addition, we exposed the results of precision and recall (Figure 3), and recall rate (Figure 4) for analyzing each iteration. After obtaining the evaluation results, we identified APIs recommended in top-20 lists over all iterations. In Table 4, we showed number of recommended APIs, their name and their frequency value (number of times the API was recommended over the five iterations).

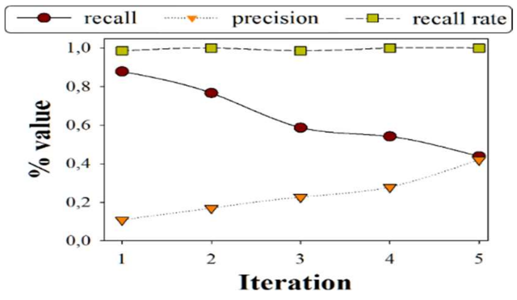

**Figure 2. Results of Evaluation Metrics for every Iteration of 5-Fold Cross Validation for Large Lists of APIs Recommended**

**Table 3. Results of Evaluation Metrics when Varying N of Top-N Lists**

| *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 |

## 5.2. API Recommendation for Game Software in Stage B

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 .xml file as relevant APIs (APIs that we expect to recommend). As that removal was made randomly, we did five replicates of API recommendation for every software of the test set in every iteration. We used CF technique along with FIS technique. Both techniques have two main attributes: i) the minimum support value; and ii) the number of NN. Thereby, we needed to find the "best" value for each attribute. In case of *minSupport*, we used the same value of 0.1 found in Stage A, since we used the same data sample, technique, and evaluation method. On the other hand, we analyzed the effect of varying *kNN*, i.e., the number of NN in top-10 lists of APIs recommended from all iterations and replicates of 5-fold cross validation method (Table 5) and we found that varying *kNN* in 20 presented the best recall value. Consequently, we chose *minSupport* = 0.1 for FIS technique and *kNN* = 20 for CF technique in Game category.

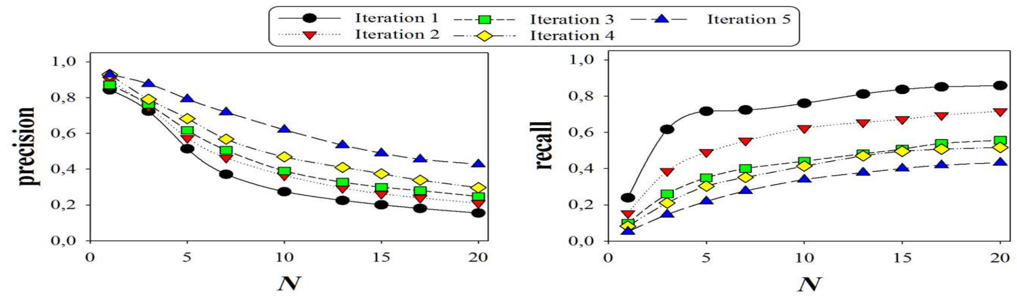

**Figure 3. Precision and Recall Metric for every Iteration of 5-Fold Cross Validation when Varying N of Top-N Lists**

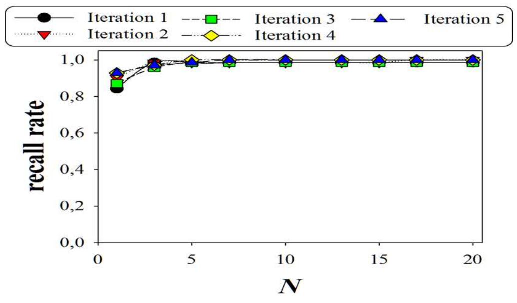

**Figure 4. Recall Rate Metric for every Iteration of 5-Fold Cross Validation when Varying N of Top-N Lists**

**Table 4. APIs Recommended in Top-20 Lists**

| # | 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 5. Effect of Varying k, i.e., Number of NN in Top-10 lists of APIs Recommended**

| *kNN* | 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 *kNN* values, we used them for evaluating our technique for recommending APIs to software engineers in Stage B when receiving large and top-*N* lists of APIs recommended. In case of top-*N*, we evaluated the effect of varying *N*. 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 (Recall = 0.643; Precision = 0.241; Recall Rate = 0.994). In addition, for analyzing every iteration, we exposed evaluation results in Figure 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 6). In addition, for analyzing every iteration, we exposed average values of precision and recall (Figure 6), and recall rate (Figure 7) for each iteration. It is important to consider that in every iteration, we did five test replicates since the removal of the 50% of APIs was randomly.

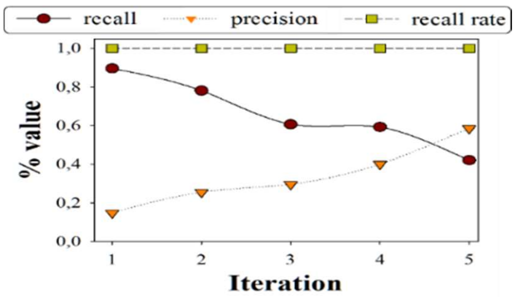

**Figure 5. Results of Evaluation Metrics for every Iteration of 5-Fold Cross Validation for Large Lists of APIs Recommended**

**Table 6. Results of Evaluation Metrics when Varying N of Top-N Lists**

| *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 |

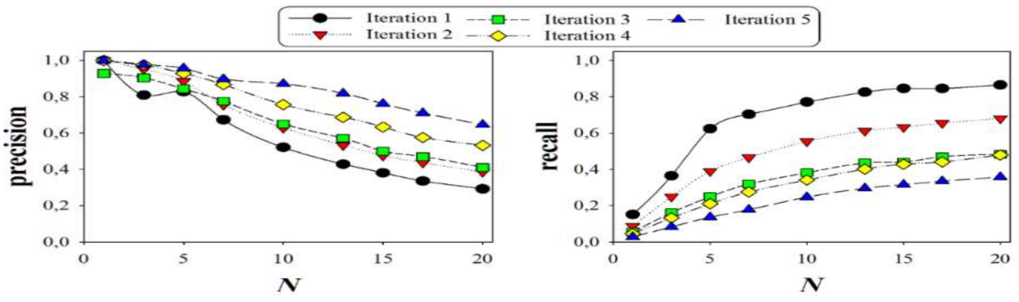

**Figure 6. Precision Metric for every Iteration of 5-Fold Cross Validation when Varying N of Top-N Lists**

After finding evaluation results, we identified APIs recommended in top-20 over all iterations. As 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. 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 7, we showed APIs and frequency value, where frequency is number of times the API was recommended in a top-20 list over the 70 *TS-software*.

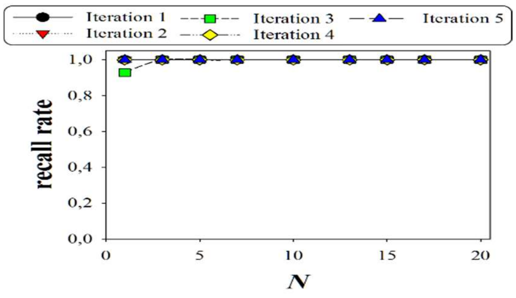

**Figure 7. Recall Rate Metric for every Iteration of 5-Fold Cross Validation when Varying N of Top-N Lists**

**Table 7. APIs Recommended in Top-20 Lists**

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

## 6. Discussion

Our study showed that our technique 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. 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.

### 6.1. API recommendation for Game software in Stage A

When analyzing recommendation results for large lists (Table 3), we expected 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 technique could correctly recommend at least one relevant API for 100% of requests.

We also exposed evaluation results for every iteration of the 5-fold cross validation regarding large lists of APIs recommended (Figure 2). 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 among 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 technique could correctly recommend at least one relevant API for 98.6% of the requests. For the remaining iterations, our technique 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 4), our recommendation technique 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 technique 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 technique could correctly recommend at least one relevant API for all the request tests (recall rate of 100%). In addition, 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 1) we could not receive all relevant APIs even in largest *N* of 20 since relevant APIs are greater than 20. In small *N* values, precision results 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 3), 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 3), we observed how these values increased as *N* increased. That result is normal and expected, since among 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 ones were requested. Instead, in top-1 lists of APIs recommended, in all iterations recall values were low (i.e., 5.2% to 23.9%), because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 4), in all iterations, when we requested for small lists of APIs, e.g., *N*=1 and *N*=3, our technique could correctly recommend at least one relevant API for more than 84.0% of the requests. For *N*=5, our technique 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.

### 6.2. API recommendation for Game software in Stage B

When analyzing our recommendation technique for large lists (Table 5), we expected low precision and maximum recall values since many irrelevant items could be recommended. As expected, we obtained precision value of 24.1% and recall value of 64.3%. Furthermore, when we requested for large lists of APIs, our technique 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 5). 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 technique for evaluating this stage of recommendation, we randomly removed half of the APIs from every target software, saving them as the relevant APIs (APIs 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, 24 to 32 APIs were recommended. In iteration 3, there were 7 to 9 relevant APIs and replicate in replicate 5, 17 to 32 APIs were recommended. In iteration 4, there were 10 to 12 relevant APIs and in replicate 2, 18 to 27 APIs were recommended. Finally, in iteration 5, there were 13 to 54 relevant APIs and in replicate 2, 16 to 24 APIs were recommended.

Regarding large lists of APIs recommended (Figure 5), 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. Thus, 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 technique could correctly recommend at least one relevant API for all the requests (100%).

Instead, analyzing our recommendation regarding small lists, i.e., varying *N* in top-*N* lists of APIs recommended (Table 6), our recommendation technique was able to put relevant APIs in high-ranking positions. When we requested recommendation lists with one API (*N*=1 or *N*=3), our technique 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 7 to 20, our recommendation technique could correctly recommend at least one relevant API for more than 99.0% of the requests. In addition, we 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 (89.4% to 26.7%) and recall to be higher (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 6), those values decremented as *N* incremented in all iterations. For example, in top-1 of recommendation of iteration 3, our technique recommended some irrelevant APIs in few recommendation request tests causing precision value of 92.9%. In addition, for the remaining iterations, our technique 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 6), we observed how those values increased as *N* increased. It is normal and expected, since among more APIs recommended, major is the chance of recommending relevant APIs. For example, in top-20 lists of APIs recommended, highest recall values were achieved in all iterations. However, iteration 5 presented significant difference regarding other iterations with recommendation lists for *N*=20. Then, that difference is normal since in software of iteration 5 there were 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, recall values were low in all iterations (2.8% to 15.1%), because we cannot expect to receive all relevant APIs when just one of them had been requested.

Regarding recall rate metric (Figure 7), in iteration 3 even when we requested for small lists of APIs, e.g., *N*=1, our recommendation technique 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 technique could correctly recommend at least one relevant API for 100% of the requests. Besides, we observed from *N*=5 to *N*=20 that our recommendation technique correctly recommended at least one relevant API for all the request in all iterations.

## 7. Related Work

Many studies used and demonstrated how frequency (popularity) helps software engineers in API issues. For example, using a technique based on the popular vote of the majority where the more people use a particular version, the higher its usage, it is recommended. In their study, authors tried to recommend or dissuade from switching library versions based on global usage history [Mileva *et al.*, 2009]. In other example, frequency of API use (popularity) helped software engineers to focus their investigative efforts on APIs that more developers have found useful in the past, rather than understand large API descriptions to find what they need [Holmes; Walker, 2007]. Thus, 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, to use effectively 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]. Then, studies have been made in order to support software engineers with APIs available through tools like LIBTIC for finding API experts [Teyton *et al.*, 2013]; Precise, an automated approach to parameter recommendation for API usage, which is able to recommend API parameters frequently used in practice; or hybrid approach for automated API recommendations [Thung *et al.*, 2013].

## 8. Final Remark

In this paper, we have proposed a novel technique for API recommendation that considers Game category from SourceForge and combines Collaborative Filtering with Frequent Item Set mining. We conducted an empirical experiment with a dataset of software from Game SourceForge category. For evaluating our API recommendation technique, we used a system-centric method and we used the 5-fold cross validation. Results demonstrated that it was possible to obtain useful lists of APIs recommended, making good recommendations even in small sizing of top-*N* lists. Results also showed that our technique partially solved the cold-start by recommending useful APIs for software engineers with Game software that used a small number of APIs or did not use any APIs at all. In Stage A, regarding large lists, averaged metric values were 66.0% of recall, 33.7% of precision and 100% of recall rate. Regarding top-20 lists, averaged metric values were 57.3% of recall, 45.4% of precision and 100% of recall rate. On the other hand, in Stage B, regarding large lists, averaged metric values were 64.3% of recall, 24.1% of precision and 99.4% of recall rate. Regarding top-20 lists, averaged metric values were 61.6% of recall, 26.7% of precision and 99.4% of recall rate.

In addition, we contributed to software engineering by proposing an API recommendation technique 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.

As future work, we plan to use other well-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 also 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 by considering their preferences and feedback. Besides, we can consider the API reputations. 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.

## References

Aarseth, E. Playing Research: Methodological Approaches to Game Analysis. In: Digital Arts & Culture Conference (p.7). Melbourne. 2003.

Acharya, M.; Xie, T.; Pei, J.; Xu, J. Mining API Patterns as Partial Orders from Source Code : From Usage Scenarios to Specifications. 2007.

Anand, D.; Bharadwaj, K. K. Utilizing Various Sparsity Measures for Enhancing Accuracy of Collaborative Recommender Systems Based on Local and Global Similarities. In: Expert Systems with Applications, 38(5), 5101-5109. 2011.

Bajracharya, S.; Ossher, J.; Lopes, C. Sourcerer: An Internet-Scale Software Repository. In: ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation. pp. 1-4. 2009.

Barbosa, Y. de A. M. Um Sistema de Recomendação de Código-Fonte para Suporte a Novatos. Universidade Federal de Pernambuco. Retrieved from http://repositorio.ufpe.br:8080/xmlui/handle/123456789/2737. 2011.

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. pp. 469-474. 2008.

Bobadilla, J.; Hernando, A.; Ortega, F.; Bernal, J. A Framework for Collaborative Filtering Recommender Systems. In: Expert Systems with Applications,38(12), pp.14609-14623. 2011.

Cremonesi, P.; Garzotto, F.; Turrin, R. User-Centric vs. System-Centric Evaluation of Recommender Systems. In: Lecture Notes in Computer Science. pp. 334-351. 2013.

Cremonesi, P.; Turrin, R.; Lentini, E.; Matteucci, M. An Evaluation Methodology for Collaborative Recommender Systems. In: International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution. pp. 224-231. 2008.

Dapeng, H.; Qianhui, L.; Jingmin, Z. An Improved Similarity Algorithm for Personalized Recommendation. In: International Forum on Computer Science-Technology and Applications. pp. 54-57. 2009.

Duala-Ekoko, E.; Robillard, M. P. Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study. In: International Conference on Software Engineering. pp. 266-276. 2012.

Holmes, R., & Walker, R. J. Informing Eclipse API production and consumption. Proceedings of Workshop on Eclipse Technology eXchange – Eclipse. pp. 70–74. 2007.

Jannach, D.; Zanker, M.; Felfernig, A.; Friedrich, G. Recommender Systems: An Introduction. 2010.

Lopes, C.; Bajracharya, S.; Ossher, J.; Baldi, P. UCI Source Code Data Sets. Retrieved January 19, 2015, from http://www.ics.uci.edu/~lopes/datasets. 2010.

Maffort, C.; Valente, M. T.; Bigonha, M.; Hora, A.; Anquetil, N.; Menezes, J. Mining Architectural Patterns Using Association Rules. In: International Conference on Software Engineering and Knowledge Engineering. pp. 375-380. 2013.

McLaughlin, M. R. M.; Herlocker, J. L. J. A Collaborative Filtering Algorithm and Evaluation Metric that Accurately Model the User Experience. In: ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 329-336. 2004.

Mileva, Y. M.; Dallmeier, V.; Burger, M.; Zeller, A. Mining Trends of Library Usage. In: Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution. pp. 57-62. 2009.

Niwattanakul, S.; Singthongchai, J.; Naenudorn, E.; Wanapu, S. Using of Jaccard Coefficient for Keywords Similarity. In: International MultiConference of Engineers and Computer Scientists. 2013.

Núnez-valdéz, E. R.; Aguilar, L. J.; Lovelle, J. M. C.; Martínez, O. S.; García-bustelo, B. C. P.; García-Diaz, V.; Montenegro-Marin, C. E.; Espada, J. P. Plataforma de Recomendación de Contenidos para Libros Electrónicos Inteligentes Basada en el Comportamiento de los Usuarios. In: Ventana Informática. 14, pp. 25-40. 2012.

Park, S.; Pennock, D.; Madani, O.; Good, N.; DeCoste, D. Naïve Filterbots for Robust Cold-Start Recommendations. In: Conference on Knowledge Discovery and Data Mining. 2006.

Robillard, M.; Walker, R.; Zimmermann, T. Recommendation Systems for Software Engineering. In: IEEE Software, 27(4), pp. 80-86. 2010.

Robillard, M.; Walker, R.; Zimmermann, T. Recommendation Systems in Software Engineering. In: IEEE Software. 2014.

Schein, A. I.; Popescul, A.; Ungar, L. H.; Pennock, D. M. Methods and Metrics for Cold-Start Recommendations. In: Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. 2002.

Son, L. H. Dealing with the New User Cold-Start Problem in Recommender Systems: A Comparative Review. In: Information Systems. 2014.

Sun, C.; Khoo, S.; Zhang, S. J. Graph-Based Detection of Library API Imitations. In: International Conference on Software Maintenance. pp. 183-192. 2011.

Teyton, C.; Falleri, J.-R.; Morandat, F.; Blanc, X. Find your Library Experts. In: Working Conference on Reverse Engineering. pp. 202-211. 2013.

The Java Tutorials. The JavaTM Tutorials. Retrieved June 20, 2014, from https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html. 2014.

Thung, F.; Lo, D.; Lawall, J. Automated Library Recommendation. In: Working Conference on Reverse Engineering. pp. 182-191. 2013.

Thung, F.; Wang, S.; Lo, D.; Lawall, J. Automatic Recommendation of API Methods from Feature Requests. In: International Conference on Automated Software Engineering. pp. 290-300. 2013.