Combining Collaborative Filtering and Topic Modeling for More Accurate Android Mobile App Library Recommendation

Huan Yu*, Xin Xia†, Xiaoqiong Zhao*, and Weiwei Qiu*
*College of Computer Science and Technology, Zhejiang University, China
†Department of Computer Science, University of British Columbia, Canada
yuhuan@zju.edu.cn, xxia02@cs.ubc.ca, zhaoxiaoqiong@zju.edu.cn, qiuweiwei@zju.edu.cn

ABSTRACT

The applying of third party libraries is an integral part of many mobile applications. With the rapid development of mobile technologies, there are many free third party libraries for developers to download and use. However, there are a large number of third party libraries which always iterate rapidly, it is hard for developers to find available libraries within them. Several previous studies have proposed approaches to recommend third party libraries, which works in the scenario where a developer knows some required libraries, and needs to find other relevant libraries with limited knowledge. In the paper, to further improve the performance of app library recommendation, we propose an approach which combines collaborative filtering and topic modeling techniques. In the collaborative filtering component, given a new app, our approach recommends libraries by using its similar apps. In the topic modelling component, our approach first extracts the topics from the textual description of mobile apps, and given a new app, our approach recommends libraries based on the libraries used by the apps which has similar topic distributions. We perform experiments on a set of 1,013 apps, and the results show that our approach improves the state-of-the-art by a substantial margin.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*;

KEYWORDS

Library Recommendation, Topic Modeling, Collaborative Filtering, Android App

ACM Reference format:
Huan Yu*, Xin Xia†, Xiaoqiong Zhao*, and Weiwei Qiu*. 2017. Combining Collaborative Filtering and Topic Modeling for More Accurate Android Mobile App Library Recommendation. In *Proceedings of Internetware'17, Shanghai, China, September 23, 2017,* 6 pages.
https://doi.org/10.1145/3131704.3131721

Internetware'17, September 23, 2017, Shanghai, China

ACM ISBN 978-1-4503-5313-7/17/09…$15.00
https://doi.org/10.1145/3131704.3131721

1 INTRODUCTION

Mobile apps continue to grow in popularity at a rapid pace, and they become one of the most popular software systems today. Mobile app markets such as Google Play has more than 2.8M apps[1]. Although the rapid development of app market has brought huge benefits, competition in the app industry is very tough as many apps provide similar functionality [20]. To reduce development time, increase the reliability of software, and improve the productivity, mobile app developers frequently use third-party libraries.

Today, with the rapid development of the mobile app industry, there have been a large amount of third-party libraries for developers to download and use. However, it is still a challenge for app developers to effectively use these libraries, since they might not be aware of these libraries as they are released. App developers might reinvent the wheel rather than reuse the suitable third-party libraries. To bridge the gap between the large amount of available third-party libraries and the developers that need to use them, in this paper, we propose an approach to automatically recommend third-party libraries.

In recent years, several research works have been conducted on third party libraries recommendation [5, 6, 16]. Association rule mining [1] is a widely used technique among those works [5, 6, 16], which identify the libraries co-occurrence relationships by mining historical mobile apps. The recommendation accuracy of association rule mining is highly influenced by the probability of the co-occurrence of third party libraries. Thung et al. proposed an approach to recommend an entire third party library, which is based on the libraries that a certain number of projects commonly uses [16] . However, the above mentioned studies do not take into account the textual descriptions of projects. In practice, description information of an app always provides useful information which includes the intent of application, the functions that application provides, the running environment, etc. Mining these textual description to further improve the performance of library recommendation for mobile apps.

In this paper, we propose an automated approach *AppLibRec* which combines Latent Dirichlet Allocation [3] and collaborative filtering [4, 9, 14] to recommends a list of third party libraries for mobile apps. Specifically, *AppLibRec* performs two kinds of analysis: README file (textual description) based analysis (*RM-based*) and libraries based analysis (*Lib-based*). In the *RM-based* component, we use LDA model which takes textual descriptions as input to compute the similarity between those textual descriptions. In the

[1] https://www.statista.com/statistics/266210

Lib-based component, we leverage collaborative filtering to recommend libraries by using its similar apps. To make a comparison, we choose LibRec proposed by Thung et al. [16] as a baseline. We perform experiments on a set of 1,013 apps, and the results show that our approach improves the precision@5 and recall@5 of LibRec by 38% and 35%, respectively.

The main contributions of this paper are:

- We propose a hybrid approach that combines topic model and collaborative filtering to recommend third party libraries for mobile apps.
- Experiments on 1,013 apps show the effectiveness of our *AppLibRec* approach, and our approach outperforms the state-of-the-art approach by a substantial margin.

2 RELATED WORK

Library Recommendation. To our best knowledge, Thung et al.'s study [16] is most related to ours. Thung et al. proposed LibRec which combines association rule mining and a nearest-neighbor-based collaborative filtering approach to recommend libraries for projects on GitHub.

Recommender Systems in SE. Recommender systems are widely utilized in software engineering [2, 8, 11, 13, 15–18, 21–23]. Many previous studies have proposed approaches to recommend various code elements (e.g., method calls, blocks of code), using various information sources (e.g., source code, commit logs) by various heuristics. Mandelin et al. [11] propose the problem of jungloid mining, jungloid mining code fragments that satisfy the query which describes the input and output types. Robbes et al. [13] improve code auto-completion by analyzing recorded program history. Serval tools provide real-time code clone detection [8, 10].

Compared with these code-level recommendation systems, there are several previous that works at a different level of granularity i.e., library-level, and recommends analogical third party libraries to the developers. Thung et al. [16] extract the co-occurrence libraries which are always commonly used together on the historical third party library. Teyton et al. [15] recommend libraries that can replace an existing library in a software project by analyzing the evolution of projects' dependencies on third party libraries.

Inspired by these studies, our approach not only relies on the information about the third party libraries that other (e.g., historical) mobile apps used, but also relies on the textual descriptions of mobile applications. Also, we focus a different problem, i.e., recommending third-party libraries for mobile apps.

3 PROBLEM DEFINITION

In this section, we first describe the process on how to identify libraries from mobile apps. Then, we present the conditions to define our problem.

3.1 Library Identification

Different from Thung et al.'s study which extract the libraries used in a Java project by analyzing its Maven configuration file. As we noticed, only 51% of our collected apps use Maven to build the system. Here, we use the following heuristics to identify libraries from Android apps:

```
package com.anysoftkeyboard;

import android.content.Intent;
import android.os.Bundle;
import android.support.annotation.NonNull;
import android.support.annotation.Nullable;
import android.support.v4.app.Fragment;

import com.anysoftkeyboard.ui.settings.MainSettingsActivity;
import com.menny.android.anysoftkeyboard.AskGradleTestRunner;
import com.menny.android.anysoftkeyboard.R;

import net.evendanan.chauffeur.lib.FragmentChauffeurActivity;
import net.evendanan.frankenrobot.Diagram;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.robolectric.Robolectric;
import org.robolectric.RuntimeEnvironment;
import org.robolectric.util.ActivityController;

/**
 * Driver for a Fragment unit-tests
 */
@RunWith(AskGradleTestRunner.class)
public abstract class RobolectricFragmentTestCase<T extends Fragment>
```

Figure 1: Sample Java Class Import of Mobile Application

Libraries in Maven repository. For the 51% apps which used Maven to build the system, we first check their pom.xml to identify the libraries, and we record the set of libraries as $Set1$.

Libraries in other apps. For the remaining 49% apps, we analyze the "import" statements in their source code files. A typical Android mobile application usually uses several third party libraries, which developers use by importing in their own java class file (see Figure 1). We model the libraries data as a forest W, every tree in the forest represents a third party library, the non-leaf nodes in the tree represent potential libraries since they might be sub packages of a library, while leaf nodes in the tree represent APIs in the sub packages of libraries.

After we analyze all the collected apps, we manually check whether the potential libraries in $Set2$ are true libraries. To do so, we manually search online by inputting the name of each of the potential library, and if we find its GitHub or SourgeForge website which clearly stated it is a third-party library, we denote it as true third-party library.

An Illustrative Example. Figure 2 illustrates the corresponding third party libraries forest extracted from Figure 1. We extract the four potential libraries {org.junit, net.evendanan.chauffeur.lib, net.evendanan.frankenrobot, org.robolectric}. These packages which are extracted as recommendatory libraries do not have leaf "sibling" nodes in our experimental dataset, i.e., all the "sibling" nodes of these sub packages are non-leaf nodes if they have "sibling" nodes.

3.2 Problem Definition

In this paper, we define the library recommendation problem as follows. The set of all available libraries in our dataset are called "LibsRepos", the set of libraries that are currently used in the mobile application are called "UsedLibs", while the set of libraries that are to be recommended are called "AppLibRecs". The goal of our approach is to find "AppLibRecs" that satisfies the following conditions:

1) $AppLibRecs \subseteq LibsRepos$
2) $AppLibRecs \cap UsedLibs = \emptyset$

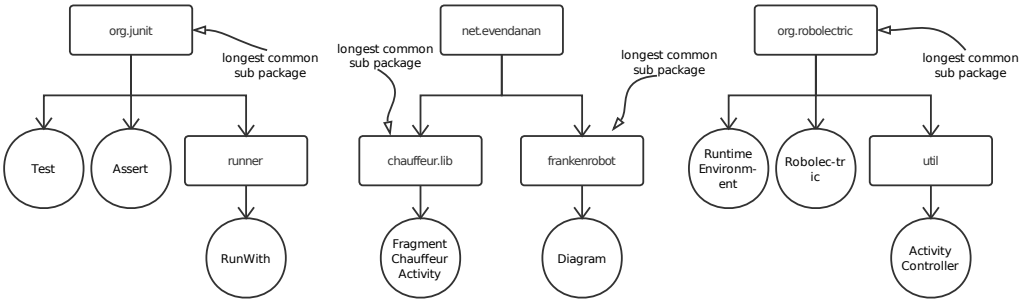


Figure 2: A Partial Libraries Forest

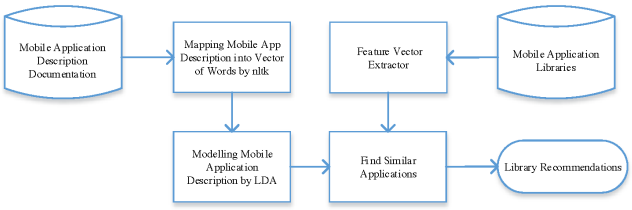


Figure 3: Third Party Libraries Recommendation Framework

4 OUR APPROACH

We first provide a description of our overall framework in Figure 3, then propose our *AppLibRec* method to solve the third party libraries recommendation problem. In section 4.2, we describe the *RM-based* component which uses LDA model. In section 4.3, we present the *Lib-based* component which uses collaborative filtering. Finally, we aggregate the two components presented in section 4.4.

4.1 Overall Framework

Figure 3 shows the overall framework of our approach also called as *AppLibRec*. It contains two major components: *RM-based* and *Lib-based*. We combine these two lists of recommendation libraries into a new list along with their final scores. We are to recommend the libraries that have highest scores.

4.2 *RM-based* Analysis

4.2.1 ***Training LDA Model***. This step is to train the LDA Model, and converts the textual descriptions of training applications to topic vectors. In natural language processing, a topic represents a distribution of words, and a document is a distribution of topics. Using LDA model can convert a textual description to a topic vector which is the document-topic distribution. After the text pre-processing (i.e., tokenization and stemming), we get a training textual description corpus. LDA takes as input that corpus and a number of parameters which are described in section 5. For each document m, LDA would generate a topic proportion vector θ_m that contains k elements. Each element presents a topic, and ranges from 0 to 1 which corresponds the proportion of the term (i.e., word) in m belonging to the topic in m. We denote the θ_m as the following :

$$\theta_m = (T[1], T[2], \ldots, T[k]) \tag{1}$$

where $T[i]$ is the proportion of ith topic in document m, $i \in \{1, 2, \ldots, k\}$, and k is the number of all topics of training applications.

4.2.2 ***Find Similar Textual Description***. Given a new application which requires to be recommended libraries, we first convert the textual description of the new application into a topic vector by using the LDA model which is trained in the Training LDA Model step. Finally, we calculate the distance between this topic vector and the topic vectors of training applications which have extracted in the first step. Cosine similarity is employed to measuring the distance.

$$Consine(New, Training) = \frac{\theta_{New} \cdot \theta_{Training}}{|\theta_{New}||\theta_{Training}|} \tag{2}$$

where $\cdot$ denotes the dot product, and $|\theta_i|$ presents the size of a topic vector.

We select the top k applications which have the highest cosine similarity scores as the k-nearest neighbors for the new project. Finally, we collect all of the libraries that are used by k-nearest neighbors and we are to calculate the score for each library in these libraries. For instance, we calculate the library L score as follows:

$$Score_{RM-based}(L) = \frac{Count(L)}{k} \tag{3}$$

Where $Count(L)$ is the number of nearest neighbors that have used the library L and k is the number of nearest neighbors. The $Score_{RM-based}(L)$ ranges from 0 to 1. The libraries that have the highest scores might be recommended for developers.

4.3 *Lib-based* Analysis

4.3.1 ***Feature Vector Extractor***. This step is to convert the set of libraries used by training applications to feature vectors. We denote the set of all libraries which are arranged in alphabetical order of their name as F, and the index of each library is a unique $F[i]$, where $i \in \{1, 2, \ldots, n\}$ and the n is the number of all distinct libraries. The feature vector of application a, is defined as follows:

$$Vector(a) = (index(F[0], a), index(F[1], a), \ldots, index(F[n], a)) \tag{4}$$

where $F[i] = 1$ if application a uses the ith library, or $F[i] = 0$ otherwise, $i \in \{1, 2, \ldots, n\}$, and n is the number of all distinct libraries.

4.3.2 ***Find Similar Applications.*** Given a new application which requires to be recommended libraries, we first convert the set of libraries that the new application has used into a feature vector in the same processing as was done in the Feature Vector Extractor step. Then it calculates the distance between this feature vector and each feature vector of all training applications which have extracted in the first step (i.e., Feature Vector Extractor). PCC is employed to measuring the distance [12], [7].

$$Sim(New, Training) = \frac{\sum_{i \in I_N \cap I_T} (F_{N,i} - \overline{F}_N)(F_{T,i} - \overline{F}_T)}{\sqrt{\sum_{i \in I_N \cap I_T} (F_{N,i} - \overline{F}_N)^2} \sqrt{\sum_{i \in I_N \cap I_T} (F_{T,i} - \overline{F}_T)^2}} \quad (5)$$

where $I_N \cap I_T$ is the set of libraries invoked by both project New and project $Training$, $F_{N,i}$ is the library extracted by project New, $F_{N,i}$ = 1 if application New uses the ith library, or $F_{N,i}$ = 0 otherwise. $\overline{F}_N$ denotes the average quantity of libraries invoked by project New.

We select the top n applications which have the highest PCC scores as the n-nearest neighbors for the new project.Finally, we collect all of the libraries that are used by n-nearest neighbors and we are to calculate the score for each library in these libraries. For instance, we calculate the library L score as: $Score_{Lib-based}(L) = \frac{Count(L)}{n}$, where $Count(L)$ is the number of nearest neighbors that have used the library L and n is the number of nearest neighbors. The $Score_{Lib-based}(L)$ ranges from 0 to 1. The libraries that have the highest scores are to be recommended for developers.

4.4 Aggregator Components

In this section, We are to get the recommendation score by combining $Score_{RM-based}$ and $score_{Lib-based}$, denoted as $Score_{AppLibRec}$. For each library, the recommendation score is defined as follows:

$$Score_{AppLibRec} = \alpha \times Score_{RM-base} + \beta \times Score_{Lib-based} \quad (6)$$

where α and β represent the weights of *RM-based* and *Lib-based* respectively. We set $\alpha + \beta = 1$, so $Score_{AppLibRec}$ ranges from 0 to 1. The top n libraries which have the highest score are to recommended to developers.

5 EXPERIMENT RESULTS

In this section, we describe our dataset, and present our evaluation measures. And then, we demonstrate our research questions and the results of our experiments. Finally, we discuss the threads to validate.

5.1 Experimental Setup

We conduct experiments on real world mobile application. We first randomly download 3,117 mobile applications from GitHub, and then filtered those applications by the following criteria:

After filtering applications that not have README file, use less than 4 libraries, and are forked from other applications. There are left with 1013 applications, including popular projects such as AdAway (303.0 kLOC) which is an open source ad blocker for Android using the hosts file, NewsBlur (419.4 kLOC) which is a personal news reader.

Evaluation Metrics. Suppose that there are m applications, for each application a_i, let ground truth be the set of libraries GT_i. We recommend the set of top-k libraries P_k for a_i according to our approach. In the set of all recommendations P_k (for all projects), that includes at least one library in the ground truth (defined in Section 3). The Precision@k and Recall@k for the m applications are defined as:

$$Precision@k = \frac{1}{m} \sum_{i=1}^{m} \frac{|P_k \cap GT_i|}{|P_i|} \quad (7)$$

$$Recall@k = \frac{1}{m} \sum_{i=1}^{m} \frac{|P_k \cap GT_i|}{|GT_i|} \quad (8)$$

Ten-fold Cross Validation. Ten-fold cross validation is employed to measuring the accuracy of our method. Each fold includes nine parts of the dataset as the training data and the remaining part as the testing data. For each testing application, we drop half of the libraries as the ground truth. The remaining libraries are taken as inputs to our *Lib-based* component.

Our approach *AppLibRec* combines *RM-based* component and *Lib-based* component, that takes a number of parameters. The *RM-based* component uses LDA that accepts five parameters. We set the maximum number of iterations to 1500 and the hyperparameters α and β to $50/k$ and 0.01, where k is the number of topics. We use percentages of distinct terms in out training data rather than a fix number to set the number of topic. We vary the number of topics for 1% to 11% of the number of distinct words in our training data which is proposed by [19], and we set k = 1% by default. As the *RM-based* component would find its k-nearest neighbors, we set the number of neighbors to 30 by default. We use Python Software Foundation [2] as the LDA implementation, which uses collapsed Gibbs sampling. The *Lib-based* component uses collaboration filtering, and we set the number of neighbors to 30 by default. The other parameters in equation 6 to their default values i.e., α = 0.3 and β = 0.7. For LibRec [16], there are five parameters, and we set the parameters as follows: $minsup$ = 0.1, $minconf$ = 0.8, n(the number of neighbors)=20, α = 0.5, and β = 0.5. These settings have been show to result in the best performance.

5.2 Research Questions

We are interested to answer the following research questions:

RQ1 How effective is our proposed *AppLibRec*? How much improvement could our proposed approach gain over the state-of-the-art approach by [16]?

Our *AppLibRec* combines *RM-based* component which uses LDA model and *Lib-based* component which employs the collaboration filtering approach. In our *Lib-based* component, PCC is employed as metric to measure the distance between two applications. The state-of-the-art work introduced by [16] is a combination of association rule mining (i.e., *Rule*) and collaboration filtering which uses cosine similarity as the metric to compute this distance. To answer this question, we compare our approach with the following baselines:

1. Association Rule Mining (Rule): This method recommends libraries by mining library usage patterns expressed as association rules.

[2] https://pypi.python.org/pypi/lda

Table 1: Performance Comparison by p(Precision)

Precision	*AppLibRec*	*LibRec*		*Rule*		*CF_cosine*		LDA(*RM-based*)		CF_pcc(*Lib-based*)	
	p	p	improve	p	improve	p	improve	p	improve	p	improve
@1	0.569	0.501	**13.57%**	0.071	**701.41%**	0.491	**15.89%**	0.516	**10.27%**	0.562	**1.25%**
@3	0.386	0.278	**38.85%**	0.068	**467.65%**	0.276	**39.86%**	0.328	**17.68%**	0.378	**2.12%**
@5	0.307	0.223	**37.67%**	0.065	**372.31%**	0.214	**43.46%**	0.257	**19.46%**	0.303	**1.32%**
@7	0.256	0.186	**37.63%**	0.061	**319.67%**	0.181	**41.44%**	0.215	**19.07%**	0.251	**1.99%**
@10	0.204	0.153	**33.33%**	0.057	**257.89%**	0.146	**39.73%**	0.173	**17.92%**	0.200	**2.00%**
@15	0.153	0.114	**34.21%**	0.052	**194.23%**	0.109	**40.37%**	0.131	**16.79%**	0.151	**1.32%**
@20	0.124	0.088	**40.91%**	0.048	**158.33%**	0.087	**42.53%**	0.107	**15.89%**	0.122	**1.64%**

Table 2: Performance Comparison by r(Recall)

Recall	*AppLibRec*	LibRec		Rule		CF_cosine		LDA(*RM-based*)		CF_pcc(*Lib-based*)	
	r	r	improve	r	improve	r	improve	r	improve	r	improve
@1	0.147	0.128	**14.84%**	0.026	**465.38%**	0.120	**18.37%**	0.132	**11.36%**	0.143	**2.80%**
@3	0.270	0.201	**34.33%**	0.031	**770.97%**	0.196	**27.41%**	0.230	**17.39%**	0.263	**2.66%**
@5	0.337	0.249	**35.34%**	0.037	**810.81%**	0.244	**27.60%**	0.293	**15.02%**	0.334	**0.90%**
@7	0.382	0.231	**31.27%**	0.040	**855.00%**	0.281	**26.44%**	0.334	**14.37%**	0.376	**1.60%**
@10	0.429	0.291	**30.79%**	0.043	**897.67%**	0.323	**24.71%**	0.375	**14.40%**	0.422	**1.66%**
@15	0.478	0.328	**31.68%**	0.045	**962.22%**	0.358	**25.10%**	0.421	**13.54%**	0.470	**1.70%**
@20	0.508	0.363	**31.61%**	0.052	**876.92%**	0.378	**25.59%**	0.451	**12.64%**	0.501	**1.40%**

2. Collaboration Filtering with Cosine Simiarity (CF_cosine): This method recommends libraries by investigating the set of libraries that are used by similar applications, using a nearest neighbor based collaborative filtering approach. It uses cosine similarity as the metric to compute this distance.

3. LibRec: Thung et al. proposed LibRec which combines association rule mining and collaborative filtering with cosine similarity to recommend libraries for projects on GitHub.

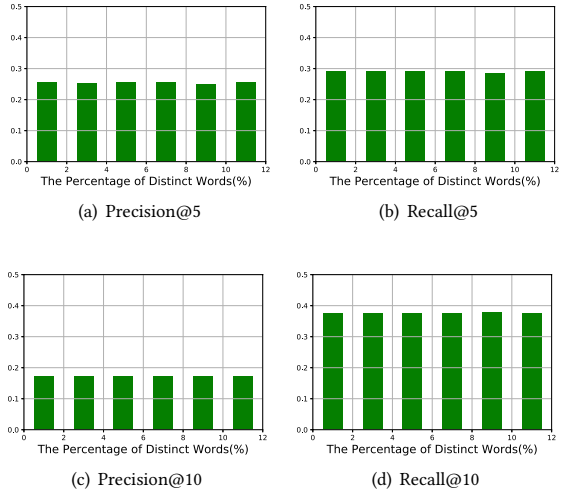
4. LDA: LDA is commonly used latent factor based model for text similarity. Our *RM-based* component recommends libraries by finding similar applications, using LDA model to find the similar applications.

5. Collaboration Filtering with PCC (CF_pcc): Our *Lib-based* component recommends libraries by investigating the set of libraries that are used by similar applications, also using a nearest neighbor based collaborative filtering approach. It uses PCC as the metric to compute this distance.

We tune each algorithm to its best parameter, and the Table 1 and Table 2 present the precision@1, precision@3, precision@5, precision@7, precision@10, precision@15, precision@20, recall@1, recall@3, recall@5, recall@7, recall@10, recall15 and recall@20 of *AppLibRec* compared with the state-of-the-art work [16] (i.e., Rule+CF_cosine) and the improvement of *AppLibRec* over it. The statistically significant improvements are highlighted in bold. From Table 1, the improvement of our approach over it is significant. *AppLibRec* outperforms it by 13.57%, 38.85%, 37.67%, 37.63%, 33.33%, 34.21% and 40.91% for average precision@1, precision@3, precision@5, precision@7, precision@10, precision@15 and precision@20. From Table 2, *AppLibRec* outperforms it by 14.84%, 34.33%, 35.34%, 31.27%, 30.79%, 31.68% and 31.61% for average recall@1, recall@3, recal@5, recall@7, recall@10, recall@15, recall@20, respectively.

RQ2 What is the performance of the *RM-based* component and *Lib-based* component?

To answer this research question, we investigate the performance of two component of *AppLibRec* separately. The result is presented in Table 1 and Table 2. Table 1 shows that *AppLibRec* outperforms the *RM-based* (i.e., LDA) component by 10.27%, 17.68%, 19.46%, 19.07%, 17.92%, 16.79% and 15.89% for precision@1, precision@3, precision@5, precision@7, precision@10, precision@15 and precision@20. Table 1 shows that *AppLibRec* outperforms the *Lib-based* (i.e., CF_pcc) component by 1.25%, 2.12%, 1.32%, 1.99%, 2.00%, 1.32% and 1.64% for average precision@1, precision@3, precision@5, precision@7, precision@10, precision@15 and precision@20. From Table 2, *AppLibRec* outperforms *RM-based* component by 11.36%, 17.39%, 15.02%, 14.37%, 14.40%, 13.54% and 12.64% for average recall@1, recall@3, recall@5, recall@7, recall@10, recall15 and recall@20, respectively. And *AppLibRec* outperforms *Lib-based* component by 2.80%, 2.66%, 0.90%, 1.60%, 1.66%, 1.70% and 1.40%for average recall@1, recall@3, recall@5, recall@7, recall@10, recall15 and recall@20, respectively. The results show that it is beneficial to combine the *RM-based* and *Lib-based* components, as it improves accuracy.



(a) Precision@5 (b) Recall@5

(c) Precision@10 (d) Recall@10

Figure 4: Recall@5, Recall@10, Precision@5, and Precision@10 for Different Numbers of Topics of *RM-based* (1-11% of The Number of Distinct Words in The Training Data)

RQ3 What is the effect of varying the number of topics to the performance of *AppLibRec*?

We next investigate the effect of varying the number of topics in LDA. We vary the number of topics for 1% to 11% of the number of distinct terms in our training data [19]. As shown in Figure 4, it presents the precision@5, precision@10, recall@5 and recall@10 of *AppLibRec* for different numbers of topics. The result can be seen that the performance of *AppLibRec* over the various numbers of

topics only varies slightly, so our *AppLibRec* is stable to different number of topics that in a reasonable range.

6 CONCLUSION

The use of third party libraries allows the developer to write less code and to focus on the key business logic of their applications. The previous studies have proposed approaches to recommend third party libraries methods which are based on the libraries that an application has used. Amst them, association rule mining is widely used. However, these mobile applications may not contain the sufficient co-occurrence of third party libraries.ongIn this paper, we propose a new method *AppLibRec* to automatically recommend third party libraries for the developers. *AppLibRec* combines Latent Dirichlet Allocation and collaborative filtering to automatically recommend libraries for developers.

Acknowledgements. This research is supported by National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

REFERENCES

[1] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *VLDB*, vol. 1215, 1994, pp. 487–499.
[2] L. Bao, D. Lo, X. Xia, and S. Li, "Automated android application permission recommendation," *Science China Information Sciences*, vol. 60, no. 9, p. 092110, 2017.
[3] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
[4] R. Burke, "Hybrid recommender systems: Survey and experiments," *User modeling and user-adapted interaction*, vol. 12, no. 4, pp. 331–370, 2002.
[5] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions–incorporating relational and categorical knowledge into word embedding," in *SANER*, 2016, pp. 338–348.
[6] C. Chen and Z. Xing, "Similartech: automatically recommend analogical libraries across different programming languages," in *ASE*. ACM, 2016, pp. 834–839.
[7] A. Gayen, "The frequency distribution of the product-moment correlation coefficient in random samples of any size drawn from non-normal universes," *Biometrika*, vol. 38, no. 1/2, pp. 219–247, 1951.
[8] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "Shinobi: A tool for automatic code clone detection in the ide," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 313–314.
[9] Y. Koren, "Factor in the neighbors: Scalable and accurate collaborative filtering," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 4, no. 1, p. 1, 2010.
[10] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Instant code clone search," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 167–176.
[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 48–61.
[12] K. Pearson, "Note on regression and inheritance in the case of two parents," *Proceedings of the Royal Society of London*, vol. 58, pp. 240–242, 1895.
[13] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.
[14] L. Terveen and W. Hill, "Beyond recommender systems: Helping people help each other," *HCI in the New Millennium*, vol. 1, pp. 487–509, 2001.
[15] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 289–298.
[16] F. Thung, L. David, and J. Lawall, "Automated library recommendation," in *2013 20th Working Conference on Reverse Engineering (WCRE 2013): Proceedings: Koblenz, Germany, 14-17 October 2013*, 2013, pp. 182–191.
[17] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 272–297, 2017.
[18] X. Xia, D. Lo, X. Wang, and X. Yang, "Who should review this change?: Putting text and file location analyses together for more accurate recommendations," in *ICSME*. IEEE, 2015, pp. 261–270.
[19] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.
[20] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *ESEM*. ACM, 2016, p. 29.
[21] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot - automated generation of answer summary to developersąŕ technical questions," in *ASE 2017, to appear*.
[22] B. Xu, Z. Xing, X. Xia, D. Lo, Q. Wang, and S. Li, "Domain-specific cross-language relevant question retrieval," in *MSR*. ACM, 2016, pp. 413–424.
[23] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia, "Bug report enrichment with application of automated fixer recommendation," in *ICPC*. IEEE Press, 2017, pp. 230–240.