Diversified Third-Party Library Prediction for Mobile App Development

Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, Yun Yang

Abstract—The rapid growth of mobile apps has significantly promoted the use of third-party libraries in mobile app development. However, mobile app developers are now facing the challenge of finding useful third-party libraries for improving their apps, e.g., to enhance user interfaces, to add social features, etc. An effective approach is to leverage collaborative filtering (CF) to predict useful third-party libraries for developers. We employed Matrix Factorization (MF) approaches - the classic CF-based prediction approaches - to make the predictions based on a total of 31,432 Android apps from Google Play. However, our investigation shows that there is a significant lack of diversity in the prediction results - a small fraction of popular third-party libraries dominate the prediction results while most other libraries are ill-served. The low diversity in the prediction results limits the usefulness of the prediction because it lacks novelty and serendipity which are much appreciated by mobile app developers. In order to increase the diversity in the prediction results, we designed an innovative MF-based approach, namely LibSeek, specifically for predicting useful third-party libraries for mobile apps. It employs an adaptive weighting mechanism to neutralize the bias caused by the popularity of third-party libraries. In addition, it introduces neighborhood information, i.e., information about similar apps and similar third-party libraries, to personalize the predictions for individual apps. The experimental results show that LibSeek can significantly diversify the prediction results, and in the meantime, increase the prediction accuracy.

Index Terms—Third-party library, prediction, mobile app development, matrix factorization, diversity, accuracy bias.

1 Introduction

The rapid growth of mobile apps (referred to as *apps* hereafter) has significantly fueled the competition among app developers and app vendors. As reported by AppBrain,[1] more than 2.73 million Android apps are available on Google Play in Q2, 2019. App developers often need to enhance or upgrade their apps as soon as possible to survive or gain an advantage in the market competition. Third-party libraries (referred to as *TPLs* hereafter) offer app developers a large variety of options for improving their apps, e.g., adding social features, enhancing user interfaces, etc., as well as saving time and effort. Thus, they are used more and more widely for app development [1]. Our investigation over 61,700 Android apps on Google Play shows that 92.25% of these apps use 5 TPLs or more, with an average of 11.81. Fig. 1 shows the results of our investigation in detail. The popularity of TPLs is also evidenced by the 6,200 TPLs hosted in the repository of Android Arsenal,[2] an Android developer portal.

The large number and variety of available libraries offer a wide range of possibilities for app developers. However, new challenges are also raised for app developers. First, given the severe time-to-market constraints, potentially useful libraries often slip under the radar due to the lack of effective approaches for finding useful libraries. It is impractical for app developers to manually inspect every potential library and determine its usefulness from different perspectives, e.g., its performance, interface, etc. Thus, finding useful libraries is often impossible due to the time limitations in the competitive app markets. Second, sometimes multiple libraries are used to collectively perform one specific function. Finding the synergy between different libraries is another time-consuming process to find appropriate combinations of libraries merely through inspecting official manuals or online user reviews.

In recent years, more and more attention has been paid to the use of libraries in mobile app development. Researchers have proposed several tools for analyzing or detecting libraries used in apps, such as LibRadar [2], LibD [3], LibPecker [4] and LibScout [5]. These tools can be employed to collect TPL usage records. Derr et al. [1] conducted a

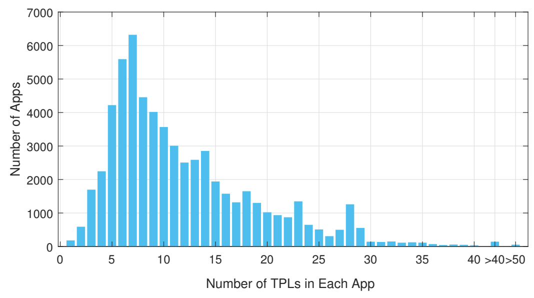


Fig. 1. Use of third-party libraries by apps

- *Qiang He and Bo Li are with the School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, AU, 3122. E-mail: {qhe, boli}@swin.edu.au*
- *Feifei Chen is with the School of Information Technology, Deakin University, Melbourne, AU, 3125. E-mail: feifei.chen@deakin.edu.au*
- *John Grundy and Xin Xia are with the Faculty of Information Technology, Monash University, Melbourne, AU, 3800. E-mail: {john.grundy, xin.xia}@monash.edu*
- *Yun Yang is with the School of Computer Science and Technology, Anhui University, PRC and the School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, AU, 3122. E-mail: yyang@swin.edu.au*

Manuscript received xxx xx, 2019; revised xxx xx, 2019.

1. https://www.appbrain.com/stats/number-of-android-apps
2. https://android-arsenal.com/

comprehensive developer survey and disclosed the needs of as well as the obstacles to the use of TPLs in Android app development. One of the major obstacles is the difficulty in finding useful TPLs.

In our preliminary study, we collect a total of 61,722 apps and investigate the TPLs used in those apps. We find that the use of TPLs has certain patterns. For example, most of the apps that use the *Facebook* library also use the *Picasso* library. The former can be used in apps to communicate with the popular *Facebook* social network and the latter can be used to download and cache images and videos from specific web servers. The two libraries may be used together by apps to perform a social media based task. This indicates that the combination of the *Facebook* library and the *Picasso* library is suitable for adding social functions to apps as many popular apps have benefited from it. Thus, apps using one of the two libraries have a good chance of finding the other one useful. The TPL usage patterns contain both explicit and implicit information that can be leveraged to facilitate predictions of useful libraries for app developers. This finding inspires a new way to predict ***useful*** libraries for apps based on collaborative filtering (CF).

Collaborative filtering has been widely employed to make predictions based on user-item interactions in a variety of domains [6] [7], including software engineering [8] [9]. Its basic philosophy is that *if two users (apps) u and v have similar behaviors (use of libraries), they will act on other items (libraries) similarly* [10]. Existing CF-based prediction methods can be categorized into memory-based methods and model-based methods [11]. The memory-based methods try to find a user's similar users and/or an item's similar items for making predictions. They can be further categorized into user-based approaches [12], item-based approaches [13] and their fusion [14]. The model-based methods train a factor model with existing user-item interactions and then make predictions. Examples of model-based methods include the clustering model [15], the aspect model [16], etc. Popularized by the Netflix Prize,[3] Matrix Factorization (MF) [17] has become the *de facto* model-based prediction approach for making predictions in recent years.

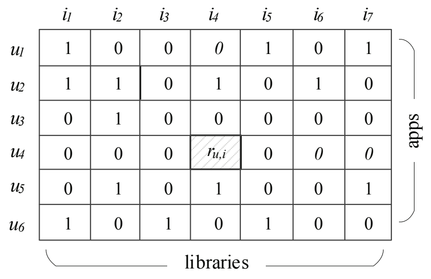
Given a user-item matrix, MF can project users, items and their interactions into a shared latent space, in which a user or an item is represented with a vector of latent features. Then, it models a user's interaction on an item as the inner product of their latent vectors. MF is capable of discovering the latent features underlying the interactions between users (apps) and items (libraries). It is more effective than these memory-based approaches when the information about user-item interactions is limited, usually indicated by a sparse user-item matrix [17]. Most of the recent recommendation systems based on matrix factorization have illustrated its effectiveness in a variety of applications [18]. In addition, our investigation shows that apps and their use of libraries result in a very sparse app-library matrix with a 1.42% sparsity. Thus, in this research, we employ MF to predict the potentially useful libraries for app developers.

To facilitate this research, we built a dataset named MALib that contains 61,722 mobile apps on Google Play and 827 libraries used in those apps across 725,502 app-library usage records - one app-library usage record indicates the use of one library in one app. Following the idea of Thung et al. [19], we selected 31,432 apps which tend to use TPLs during their development - those that are already using 10 or more TPLs [19]. We then ran three representative MF-based approaches, including SGD [20], ALS [21] and BPR [22] on the dataset to make TPL recommendations for apps. After investigating the recommendation results, we identified a critical issue, referred to as the *popularity bias* issue in this research. **A small fraction of popular libraries - those that are used by a large number of apps - dominate the prediction results and most other libraries are *ill-served* [18].** For instance, when we remove 1 library and recommend 10 libraries for each testing app with the SGD approach, an MF-based prediction approach which employs the stochastic gradient descent as the training algorithm [20], the top 8 most popular TPLs - 1% of all the libraries, contribute to 32.12% of the recommendation results. We investigated the recommendation results and MALib further and found that this was caused by the *long-tail* distribution [23] of library usage in MALib, i.e., **1% of the libraries are involved in 29.91% of app-library usage records**, as illustrated in Fig. 4.

Predicting popular libraries is trivial and does not bring much benefit to app developers. Most app developers may already be aware of those very popular libraries and have decided whether they are useful for their apps. Even if they are not, the identification of a limited number of popular libraries is not difficult using online library repositories. Most, if not all, app developers would appreciate a certain degree of novelty and serendipity in the library recommendation results, similar to many other applications as suggested by a recent study of recommender systems [24]. To address the popularity bias issue, the prediction results need to be diversified. To achieve this objective, this paper proposes a new approach named LibSeek. It employs an adaptive weighting mechanism that neutralizes the bias caused by the popularity of libraries. In addition, it introduces neighborhood information, i.e., information about similar apps and similar libraries, into MF to further tackle the popularity bias. **LibSeek can help app developers find various libraries that are potentially useful for developing or updating their apps.** To the best of our knowledge, it is the first attempt to employ MF to predict useful third-party libraries for mobile app development. It is also the first attempt to tickle the popularity bias problem relevant to the third-party library prediction. The major contributions of this paper include:

- We analyze mobile apps on Google Play and validate the idea of predicting useful libraries by running some classic MF-based prediction approaches on 31,423 mobile apps. Then, we identify the popularity bias issue, which largely reduces the significance of the prediction results;
- We propose LibSeek, a new MF-based approach that neutralizes popularity bias with an adaptive weighting mechanism and neighborhood information, significantly diversifying the prediction results;
- We conduct extensive experiments to validate the

3. https://www.netflixprize.com/



$$\begin{array}{c|ccccccc}
 & i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 \\
\hline
u_1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
u_2 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
u_3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
u_4 & 0 & 0 & 0 & r_{u,i} & 0 & 0 & 0 \\
u_5 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
u_6 & 1 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}$$

Fig. 2. An example app-library relation matrix

performance of LibSeek. Both the MALib dataset [4] and the prototype of LibSeek [5] are published online for validation and reproduction of our experimental results.

The remainder of this paper is organized as follows. Section 2 motivates our research with an analysis of the MALib dataset and the preliminary results achieved with three representative MF-based prediction approaches. Section 3 briefly introduces the main idea of MF-based predictions. Section 4 discusses LibSeek with a focus on how it addresses the popularity bias issue. Section 5 reports the results of the experiments conducted on LibSeek. Section 6 reviews related works. Finally, Section 7 concludes this paper and points out future work.

2 Motivation

In this section, we first explain the basic concept of library prediction. Then, we discuss the methodology of our preliminary experiments. Finally, we illustrate the popularity bias identified in our preliminary experiments on the MALib dataset with classic MF-based approaches.

2.1 Library Prediction

Each mobile app uses a certain number of TPLs to serve specific purposes, e.g., to enhance its user interface, to add social features, etc. We employ a $m \times n$ app-library matrix (referred to as *matrix M* hereafter) to represent the relationship between apps and libraries, assuming that there are m apps and n libraries totally in MALib. In M, the uth ($1 \leq u \leq m$) row represents whether app u uses each of the n libraries, and the ith ($1 \leq i \leq n$) column represents whether library i is used in each of the apps. An entry $r_{u,i}$ is set to 1 if app u uses library i. Otherwise, $r_{u,i}$ is set to 0. Fig. 2 demonstrates an example of M with 6 apps and 7 libraries.

Let A and L be the set of all apps and the set of all libraries in M, respectively. Library prediction aims, for each app $u \in A$, to select k libraries $\{i_1, i_2, \cdots, i_k\} \in L$ that are not used by u but are most likely to be useful to u. Then developers can prioritize the investigation of these libraries to save their effort in seeking new libraries. There are many ways to make predictions based on M. In recent years, MF-based prediction approaches have proven to be the most effective and efficient ones in many application areas [17]. They are thus employed in this research.

4. https://github.com/malibdata/MALib-Dataset
5. https://github.com/libseek/LibSeek

2.2 Preliminary Experiments

To validate the idea of TPL recommendation, we performed experiments on the MALib dataset. To recommend TPLs for apps that tend to use TPLs, we selected apps that use 10 or more TPLs for conducting the experiments, similar to the methodology employed in Thung et al.'s work [19]. More details of the MALib dataset will be discussed in Section 5.1. Based on the MALib dataset, we built a 31,423 $\times$ 752 matrix M which has 537,011 1s and 23,099,853 0s.

To simulate realistic prediction scenarios, we employed cross-validation technique [25] to set up the preliminary experiments. Cross-validation has been widely used in experiments on software-relevant predictions [8], [26], [27], [28]. The main idea is to find out whether a given approach can find libraries that are definitely useful for a given app. We conducted the preliminary experiments as follows. For each testing app, we produced a to-be-improved version of the app by randomly removing some of the libraries used by the app. Accordingly, in the app's row in M, we changed the entries corresponding to the removed libraries from 1 to 0. Then, we ran SGD [20], ALS [21] and BPR [22], three representative state-of-the-art MF-based prediction approaches, to recommend three lists of libraries for the testing app. Finally, we evaluated the prediction performance by inspecting whether the *correct* libraries, i.e., the removed libraries, are including in recommendation results.

The results were encouraging. For each app in MALib, we removed 1, 3, 5 TPLs, respectively, and recommended 5, 10 TPLs, respectively. Then, we calculated the average recall across the six cases. The three approaches were capable of finding the correct libraries with an average recall of 37.98% across all cases. This shows that a large portion of the libraries recommended by those approaches was truly useful for the corresponding apps. More details can be found in Section 5.4. This validated our idea of predicting useful libraries for apps with MF-based approaches.

2.3 Popularity Bias

However, after carefully inspecting the results of the preliminary experiments, we identify the *popularity bias issue* - a small fraction of *popular libraries* dominates the prediction results while most other libraries are *ill-served*. Unpopular libraries rarely appear in the recommendation results. Most of them are not recommended at all. Fig. 3 illustrates the prediction results achieved by SGD in one set of experiments on 31,432 apps. In this set of experiments, for each app, we randomly removed 1 library and produced a recommendation list with 10 libraries. In Fig. 3, for each removed library, the total number of times it is removed is indicated by a corresponding blue star - these removed libraries are the correct libraries that are expected to be included in the prediction lists. For each remaining library, the total number of times that the library appears in all the apps is represented by a corresponding golden diamond. Given 31,432 recommendation lists, the total number of times that each library is included in these lists is represented by a corresponding red star. All libraries are given unique IDs and ordered by their popularity indicated by the magenta diamonds in Fig. 3, i.e., the popularity of each library across all the apps, with the most popular to the left.

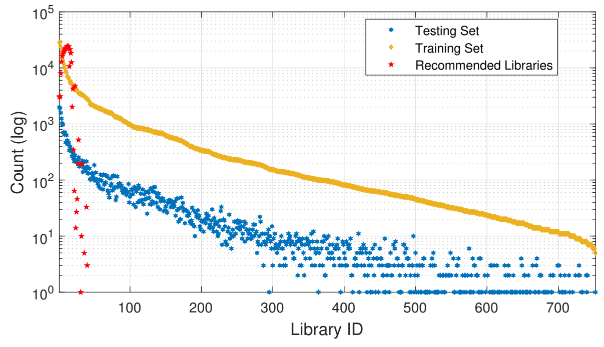


Fig. 3. Contrast between testing set and predicted libraries by MF

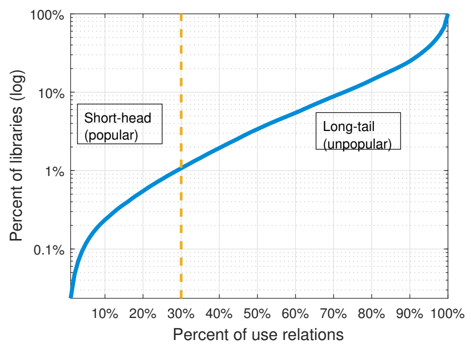


Fig. 4. Usage distribution of MALib

The red stars in Fig. 3 indicate that the recommendation results are extremely condensed in a very small fraction of popular libraries. In our preliminary experiments, we expected that the recommendation results would cover all the removed libraries. However, only 34 libraries were included in the 31,432 prediction lists. That was only 4.52% of all the 752 removed libraries. We evaluated the diversity in the recommendation results with two widely used metrics, i.e., coverage and inter-list diversity. The results were consistent with what is demonstrated in Fig. 3 - the diversity in the recommendation results was low. The details of the metrics and the evaluation results can be found in Section 5.3 and 5.4, respectively.

Recommending popular libraries does not bring much benefit, as app developers are likely to have already known, inspected or even tried out these popular libraries. In addition, such popularity-oriented recommendations are not personalized and thus are not interesting to most app developers. On the other hand, unpopular libraries are usually hard to find and easy to overlook. Recommending less popular libraries offers novelty and serendipity to app developers. Thus, it is important to diversify the library predictions for app developers. To do this, we investigated MALib and found the root cause for the low diversity in the prediction results obtained by the conventional MF-based prediction approaches in the preliminary experiments. As demonstrated by the blue stars and golden diamonds in Fig. 3, the use of libraries in apps follows the *long-tail* distribution [23], where the majority of library usages are condensed in a small fraction of popular libraries. Fig. 4 demonstrates the usage of libraries in the apps in MALib and shows the high skewness of the prediction results. Specifically, the top 1% most popular libraries were involved in approximately 29.91% of the app-library usage records in MALib. Conventional MF prediction approaches train the prediction models with the known entries in M, i.e., existing use of libraries in apps. Thus, they tend to include too many popular libraries in their prediction results. This issue is so critical that it led to a *no-tail* distribution of libraries in the prediction results, as shown by the red stars in Fig. 3. Thus, the prediction results must be diversified.

3 Matrix Factorization

Whether a library i is used by an app u is represented by entry $r_{u,i}$ in the app-library matrix. Such information is explicit. However, the reasons why a library is used by an app are usually implicit, depending on various factors, e.g., the features, interface and performance characteristics of the library. A major strength of matrix factorization is its ability to leverage both explicit and implicit information on app-library usage. Given an app-library matrix M, matrix factorization models both apps and libraries in a joint latent factor space of dimensionality f. In that joint latent factor space, app-library usage is modeled as inner products. Each library i in M is associated with a column vector $y_i \in \mathbb{R}^f$ while each app u is associated with a column vector $x_u \in \mathbb{R}^f$. Vector y_i measures how much library i possesses those f factors and vector x_u measures the extent of interest app u has in the corresponding libraries. In this way, each missing entry in M can be approximated by:

$$\hat{r}_{u,i} = x_u^T \cdot y_i \tag{1}$$

where symbol $\cdot$ represents the inner product and $\hat{r}_{u,i}$ is an approximation of the real preference $r_{u,i}$ which is app u's overall interest in library i. Thus, the entire matrix M can be approximated:

$$M \approx X^T Y \tag{2}$$

where $X = \{x_1, x_2, \cdots, x_m\} \in \mathbb{R}_{f\times m}$ and $Y = \{y_1, y_2, \cdots, y_n\} \in \mathbb{R}_{f\times n}$. X here is referred to as the app latent feature matrix and Y the library latent feature matrix.

To estimate all vectors $x_u \in \mathbb{R}^f$ and $y_i \in \mathbb{R}^f$, many of the recent matrix factorization approaches directly model the known information, i.e., entries whose value is 1 in M, and employ a regularized model to avoid over-fitting. When factorizing M, they attempt to minimize the squared loss function below:

$$\min_{X,Y} Loss = \sum_{r_{u,i}>0} (r_{u,i} - x_u^T y_i)^2 + \lambda(\|x_u\|^2 + \|y_i\|^2) \tag{3}$$

where λ is the degree of regularization. As the loss function is not convex, stochastic gradient descent (SGD) [20] or alternating least squares optimization (ALS) [21] are employed to find the approximate optimal solution by Eq. (4) and Eq. (5):

$$x_u = x_u - l_x \frac{\partial Loss}{\partial x_u} \tag{4}$$

$$x_i = y_i - l_y \frac{\partial Loss}{\partial y_i} \tag{5}$$

where l_x and l_y are the learning rates that are usually set to the same value.

TABLE 1
Summary of Notations

Symbols	Description
M	the app-library matrix
L	the full set of libraries in M
A	the full set of apps in M
u, v	apps namely u and v, respectively
i, j	libraries namely i and j, respectively
$L(u)$	all the libraries used by u
$A(i)$	all the apps using library i
$N_A(u)$	the selected neighbors of app u
$N_L(i)$	the selected neighbors of library i
N_u	matrix consists of latent factors of each neighbor app in $N_A(u)$
N_i	matrix consists of latent factors of each neighbor TPL in $N_L(i)$
X	app latent feature matrix
Y	TPL latent feature matrix
x_u	latent factor vector of app u
y_i	latent factor vector of library i
λ	regularization coefficient of MF
$r_{u,i}$	entity of M relevant to app u and TPL i
R_u	the uth row vector of M, i.e., entities relevant to app u
R_i	the ith column vector of M, i.e., entities relevant to TPL i
$w_{u,i}$	weight of entity $r_{u,i}$
W_u	diagonal matrix with all the entity weights relevant to app u
W_i	diagonal matrix with all the entity weights relevant to TPL i
$SimApp(u,v)$	similarity between apps u and v
$SimLib(i,j)$	similarity between TPLs i and j
$Sa(u,v)$	normalized similarity between apps u and v
$Sl(i,j)$	normalized Similarity between TPLs i and j
Su	vector with all similarities between app u and its neighbor apps in $N_A(u)$
Si	vector with all similarities between TPL i and its neighbor TPLs in $N_L(i)$
pn	number of TPLs in each prediction list
rm	number of TPLs removed from each testing app
k	number of apps or TPLs selected as neighbors
α	parameter controlling how much MF relies on neighborhood information
$weight$	parameter weighting explicit entries

Based on the estimated entries in M, predictions can be made whether the corresponding app would take an interest in the corresponding library. However, as illustrated and discussed in Section 2, conventional MF-based prediction approaches suffer from low diversity in their results caused by popularity bias. They highly tend to include a small fraction of the libraries that are more popular.

4 LibSeek

This section presents LibSeek, our novel MF-based approach for diversified library prediction for apps. The first means employed by LibSeek is to neutralize the skewness in app-library usage caused by popular libraries. LibSeek achieves this objective through modeling all entries in M and assigning an individualized weight to each library, giving high penalties to popular libraries. The second means is to introduce neighborhood information, information about similar apps and similar libraries, during the matrix factorization. The key notations used are shown in Table 1.

4.1 Adaptive Weighting Mechanism

As demonstrated in Section 2, the app-library matrix M uses a binary entry $r_{u,i}$ to represent the app-library usage record for app u and library i, with 1 indicating that app u uses library i and 0 otherwise. Most of the MF-based prediction approaches take only the 1 entries into account in the training process. They simply interpret a 0 entry as the user disliking the corresponding item, and thus discard all the 0 entries during the training. However, this assumption is not entirely true. For example, an entry in M being 0 can stem from various reasons. A library may have slipped under the app developer's radar when they search for useful libraries. An app developer may choose not to use a library due to, say its price or privacy policy. Thus, zero entries provide implicit information on user-item interactions [29], i.e., the use of libraries in apps in the context of this research. For ease of representation, we refer to the non-zero entries M as *explicit entries* while the zero entries as *implicit entries*. To fully leverage both the explicit and implicit information contained in M, LibSeek models not only the explicit entries but also the implicit entries to make third-party library predictions. By nature, the implicit information provided by zero entries are usually associated with low confidence [29]. Thus, LibSeek assigns to implicit entries weights that are lower than those for explicit entries.

As discussed in Section 2, the main root causes of the popularity bias in conventional MF-based predictions is the long-tail distribution of app-library usage. To address this, we introduce an adaptive weighting mechanism that adaptively assigns weights to libraries according to their popularity, giving higher weights to less popular libraries. In this way, each library is assigned a personalized weight to be used during the matrix factorization process.

LibSeek employs Eq. (6) as its loss function instead of Eq. (3):

$$\min_{X,Y} Loss = \sum_{u=1}^{m}\sum_{i=1}^{n} w_{u,i}(r_{u,i} - x_u^T y_i)^2 + \lambda\left(\sum_{u=1}^{m}\|x_u\|^2 + \sum_{i=1}^{n}\|y_i\|^2\right) \tag{6}$$

where λ is used to control the regularization. We use $w_{u,i}$ to adaptively assign different weights to different libraries in M. It is calculated based on the popularity of the corresponding library as follows:

$$w_{u,i} = \begin{cases} 1 & r_{u,i} = 0 \\ 1 + weight/(1 + \ln p_i) & r_{u,i} = 1 \end{cases} \tag{7}$$

where $weight$ is used to prioritize the non-zero entries over zero entries and p_i denotes the popularity of library i in M measured by the number of apps using i. For example, if library i is used by 1,000 apps, there is $p_i =$ 1,000. With Eq. (7), LibSeek assigns a personalized weight to each explicit entry in M but the same weight 1 to implicit entries. In Section 5, we will experimentally investigate the impact of *weight* on the performance of LibSeek.

4.2 Neighborhood Information Integration

Compared with CF-based approaches, MF-based prediction approaches utilize global information in M to train the app latent feature matrix X and library latent feature matrix Y. However, it does not utilize the *neighborhood information*, i.e., information on an app or a library's neighbors. Here, an app u's neighbors, denoted by $N_A(u)$, are the apps that are similar to u in terms of their use of libraries. A library i's neighbors, denoted by $N_L(i)$, are the libraries that are similar to i in terms of apps using them. Inspired by CF which employs the neighborhood information to make predictions, LibSeek leverages apps and libraries' neighborhood information to attack the popularity bias.

To fully utilize neighborhood information, LibSeek integrates both apps' and libraries' neighbors into the factorization of matrix M. The overall procedure consists of three phases. In the first phase, LibSeek finds the neighbors of each app u and library i. For each app u, LibSeek calculates the *library similarity* between $L(v)$ and $L(u)$ for each $v \in A$ ($v \neq u$). For each library i, LibSeek calculates the *app similarity* between $A(i)$ and $A(j)$ for each $j \in L(j \neq i)$. All entries in M are binary. Thus, LibSeek employs Eq. (8) to calculate the Jaccard Correlation Coefficient (JCC) for measuring app similarity, and Eq. (9) for library similarity. For each app u, there might be many similar apps. In the second phase, LibSeek selects the top k apps that are most similar to u as u's neighbors. Similarly, it selects the top k libraries that most similar to i as i's neighbors. In the final phase, LibSeek integrates both apps and libraries' neighborhood information into the regularization when factorizing M.

The app similarities and library similarities can be calculated offline prior to the MF and updated periodically by Eq. (8) and Eq. (9), respectively.

$$SimApp(u,v) = \frac{\sum_{l \in L} r_{u,l} \times r_{v,l}}{\sum_{l \in L} (r_{u,l} + r_{v,l} - r_{u,l} \times r_{v,l})} \tag{8}$$

$$SimLib(i,j) = \frac{\sum_{u \in A} r_{u,i} \times r_{u,j}}{\sum_{u \in A} (r_{u,i} + r_{u,j} - r_{u,i} \times r_{u,j})} \tag{9}$$

Different neighbors have different similarities, thus all similarities of selected neighbors $v \in N_A(u)$ and $k \in N_L(i)$ should be normalized as follows:

$$Sa(u,v) = \frac{SimApp(u,v)}{\sum_{r \in N_A(u)} SimApp(u,r)} \tag{10}$$

$$Sl(i,j) = \frac{SimLib(i,j)}{\sum_{k \in N_L(i)} SimLib(i,k)} \tag{11}$$

LibSeek now combines the global information and neighborhood information using Eq. (12) as the loss function:

$$\begin{aligned}\min_{X,Y,N_A,N_L} Loss = & \sum_{u=1}^{m}\sum_{i=1}^{n} w_{u,i}(r_{u,i} - x_u^T y_i)^2 \\ & + \lambda(\sum_{u=1}^{m} \|x_u\|^2 + \sum_{i=1}^{n} \|y_i\|^2) \\ & + \alpha \sum_{u=1}^{m} \sum_{v \in N_A(u)} Sa(u,v)\|x_u - x_v\|^2 \\ & + \alpha \sum_{i=1}^{n} \sum_{j \in N_L(i)} Sl(i,j)\|y_i - y_j\|^2 \end{aligned} \tag{12}$$

where parameter α determines how much the predictions rely on the neighborhood information. A higher α drives LibSeek to utilize more of the neighborhood, which will diversify the prediction results more significantly. The impact of α on the performance of LibSeek will be experimentally evaluated in Section 5. $N_A(u)$ is the set of app u's top k similar neighbors and $N_L(i)$ is the set of library i's top k similar neighbors.

To approximate the global minimum of Eq. (12), we fix the app factor vector x_u and the library factor vector y_i alternately. In this way, Eq. (12) becomes a quadratic function that can be minimized through an alternating least squares optimization process which goes through three phases [29].

In the first phase, we fix y_i in Eq. (12) and then compute the partial derivative of x_u. With y_i fixed, we could treat x_u as the only variable during the derivation process. The partial derivative of x_u is:

$$\begin{aligned}\frac{\partial Loss}{\partial x_u} = & -2\sum_{i=1}^{n} w_{u,i}(r_{u,i} - x_u^T y_i) y_i + 2\lambda x_u \\ & + 2\alpha \sum_{v \in N_A(u)} Sa(u,v)(x_u - x_v) \\ = & -2\sum_{i=1}^{n} w_{u,i} r_{u,i} y_i + 2\sum_{i=1}^{n} w_{u,i} x_u^T y_i y_i + 2\lambda x_u \\ & + 2\alpha \sum_{v \in N_A(u)} Sa(u,v)(x_u - x_v) \\ = & -2\sum_{i=1}^{n} w_{u,i} r_{u,i} y_i + 2\sum_{i=1}^{n} w_{u,i} y_i^T x_u y_i + 2\lambda x_u \\ & + 2\alpha \sum_{v \in N_A(u)} Sa(u,v) x_u - 2\alpha \sum_{v \in N_A(u)} Sa(u,v) x_v \end{aligned} \tag{13}$$

As discussed in Section 3, both x_u and y_i are latent factors and each of them has f elements, where f is the dimensionality of each latent factor. Thus, $x_u^T y_i$ is equivalent to $y_i^T x_u$.

Eq. (13) consists of 5 parts. The first part, i.e., $-2\sum_{i=1}^{n} w_{u,i} r_{u,i} y_i$, sums a total of n values of each $w_{u,i} r_{u,i} y_i$ when $i = 1, 2, \ldots, n$. We employ matrix multiplication here to describe the process. For example, as each latent factor y_i is a $f \times 1$ vector for TPL $i \in M$, all the latent factors $y_i, i = 1, 2, \ldots, n$ form the $f \times n$ matrix Y with y_i making the ith column of Y. Similarly, as each $w_{u,i}$ is a single value, we use an $n \times n$ diagonal matrix to represent a total number of n values of each weight $w_{u,i}$. Denoted by W_u, this matrix is defined as:

$$W_u(p,i) = \begin{cases} w_{u,i} & p = i, 1 \le p \le n, 1 \le i \le n \\ 0 & p \neq i, 1 \le p \le n, 1 \le i \le n \end{cases} \tag{14}$$

Each $r_{u,i}$ is also a single value in M. We define a row vector R_u with n elements to represent a total number of n values of each $r_{u,i}$. In R_u, the ith element is equivalent to $r_{u,i}$. Thus, R_u is defined as:

$$R_u = \{r_{u,1}, r_{u,2}, r_{u,3}, \cdots, r_{u,n}\} \tag{15}$$

Now, the first part of Eq. (13) can be calculated via matrix multiplication represented by $-2YW_u{R_u}^T$. The result is a column vector with f elements. Similarly, the second part of Eq. (13), i.e., $2\sum_{i=1}^{n} w_{u,i} y_i^T x_u y_i$, can be represented by $2YW_uY^Tx_u$.

Let us use a vector S_u with k elements to represent the similarities between app u and k other neighbor apps in $N_A(u)$:

$$S_u = \{Sa(u,1), Sa(u,2), \ldots, Sa(u,k)\} \tag{16}$$

and S_u^σ to denote the sum of all the elements in S_u.

For app u's neighbor app $v \in N_A(u)$, its latent factor x_v is also a $f \times 1$ latent factor. All the k neighbor apps in $N_A(u)$ form a $f \times k$ matrix, denoted as N_u and defined as:

$$N_u = \{x_1, x_2, \cdots, x_{k-1}, x_k\} \tag{17}$$

The last two parts of Eq. (13), i.e., $2\alpha\sum_{v\in N_A(u)} Sa(u,v)x_u$ and $2\alpha\sum_{v\in N_A(u)} Sa(u,v)x_v$, can also be calculated via matrix multiplication. Based on the above transformation, Eq. (13) is transformed into:

$$\begin{aligned}\frac{\partial Loss}{\partial x_u} =& -2YW_u{R_u}^T + 2YW_uY^Tx_u + 2\lambda x_u \\ &+ 2\alpha S_u^\sigma x_u - 2\alpha N_u S_u\end{aligned} \tag{18}$$

To solve x_u, we set the partial derivative to 0, i.e., to set Eq. (18) to 0. Then, we can obtain x_u as follows:

$$\begin{aligned}x_u =&(YW_uY^T + \lambda I + \alpha S_u^\sigma I)^{-1}(YW_u{R_u}^T + \alpha N_u S_u) \\ =&\left[YY^T + Y(W_u - I)Y^T + (\lambda + \alpha S_u^\sigma)I\right]^{-1} \\ &\times (YW_u{R_u}^T + \alpha N_u S_u)\end{aligned} \tag{19}$$

In the second phase, we calculate the value of y_i in a similar manner as follows:

$$\begin{aligned}y_i =&(XW_iX^T + \lambda I + \alpha S_i^\sigma I)^{-1}(XW_iR_i + \alpha N_i S_i) \\ =&\left[XX^T + X(W_i - I)X^T + (\lambda + \alpha S_i^\sigma)I\right]^{-1} \\ &\times (XW_iR_i + \alpha N_i S_i)\end{aligned} \tag{20}$$

where the $f \times m$ matrix X consists of all the latent factors of each app $u \in M$. W_i is a $m \times m$ diagonal matrix, defined as:

$$W_i(u,q) = \begin{cases} w_{u,i} & u = q, 1 \le u \le m, 1 \le q \le m \\ 0 & u \ne q, 1 \le u \le m, 1 \le q \le m \end{cases} \tag{21}$$

We use a column vector R^i to store each $r_{u,i}$ corresponding to TPL i, defined as:

$$R_i = \{r_{1,i}, r_{2,i}, r_{3,i}, \cdots, r_{m,i}\} \tag{22}$$

We use S_i to store the similarities between TPL i and its k neighbor TPLs in $N_L(i)$:

$$S_i = \{Sl(i,1), Sl(i,2), \ldots, Sl(i,k)\} \tag{23}$$

and use S_i^σ to denote the sum of all the elements in S_i.

We use a $f \times k$ matrix N_i store the latent factor vectors of all the TPLs in $N_L(i)$, defined as follows:

$$N_i = \{y_1, y_2, \cdots, y_{k-1}, y_k\} \tag{24}$$

In the third phase, by re-computing x_u and y_i alternately over a number of iterations, the loss function Eq. (12) can be minimized and finally matrix M is factorized into X and Y.

Now, M can be approximated with the inner product of X and Y, i.e., $\hat{M} \approx X \cdot Y$. Each unknown entry $r_{u,i}$ in M is approximated with a predicted value $\hat{r}_{u,i}$, which can be calculated based on the corresponding vectors x_u and y_i, i.e., $\hat{r}_{u,i} = x_u \cdot y_i$. Those unknown entries can then be used to predict useful TPLs for apps. The common practice is to sort all the unknown entries relevant to app u based on their predicted values, and then include the top k TPLs with the highest entry values in the recommendations.

4.3 Complexity Analysis

LibSeek employs both the implicit and explicit information in the app-library matrix to make recommendations. However, its computational overhead is not much higher than those that do not. Here, we theoretically analyze its complexity.

Eq. (19) and Eq. (20) show that the most time-consuming operations are the calculation of YW_uY^T and XW_iX^T, as both of them have a matrix multiplication process over three matrices. Taking Eq. (19) as an example, the result of YW_uY^T is a $f \times f$ matrix and it needs $O(n)$ time to calculate each entry in this matrix. Thus, for each app u in M, it takes $O(f^2 \times n)$ time to compute YW_uY^T. Moreover, most of the entries in W_u are 1 (corresponding to the implicit entries in M) and others (the explicit entries) have different values calculated with Eq. (7). Thus, we can replace YW_uY^T by $YY^T + Y(W_u - I)Y^T$. In this way, YY^T can be calculated offline prior to matrix factorization, which is not related to any latent factors. In addition, $(W_u - I)$ will drop all the implicit entries whose values equal to 1, and only maintain the explicit entries in $L(u)$. In this way, we can significantly reduce the time consumption of LibSeek and increase its efficiency. Let us denote the number of libraries in $L(u)$ with $C(L(u))$. LibSeek only needs $O(f^2 \times C(L(u)))$ time to compute YW_uY^T and $C(L(u)) \ll n$. Similarly, we can compute $YW_uR_u^T$ in time $O(f \times n)$ and compute N_uS_u in time $O(k \times f)$. S_u can also be calculated offline in advance. The third time-consuming operation is the matrix inversion operation in Eq. (19) and Eq. (20). In the worst-case scenario, the matrix inversion in Eq. (19) and Eq. (20) takes $O(f^3)$ time to complete as the dimension of the matrices being inverted is $f \times f$ [29].

Overall, for all the m apps in M, the total time consumed is $O(f^2 \times \mathbb{N} + f^3 \times m)$, where $\mathbb{N}$ is the number of explicit entries in M). This indicates that the running time for computing all $x_u \in X$ is linear to the number of apps m. Similarly, the running time for computing all $y_i \in Y$ is $O(f^2 \times \mathbb{N} + f^3 \times n)$. It is linear to the number of libraries n.

5 Experimental Evaluation

We evaluated the performance of LibSeek in both the diversity and accuracy of the recommendation results through comparison with 8 representative approaches, including a popularity-based approach, an association rule mining approach and 6 MF-based prediction approaches.

5.1 Dataset

We collected a total of 61,722 Android apps from Google Play via AndroidZoo[6] [30], then employed LibRadar [2] to extract TPL usage records from these Android apps. As

6. https://androzoo.uni.lu

reported by Zhang et al. [4] and Backes et al. [5], LibRadar cannot precisely find the boundary of a TPL and sometimes treats the sub-folders of a specific TPL as different TPLs. To address this issue, we manually checked and extended LibRadar's library dataset by comparing it against the TPLs available on Maven[7] and GitHub[8] and obtained a total of 898 distinct TPLs. With the updated LibRadar, we obtained 725,502 app-library usage records involving 827 distinct TPLs from these Android apps, and built the MALib dataset.

LibSeek is designed to help developers who tend to use TPLs to find useful TPLs for their mobile apps. Following the same methodology of Thung et al. [19], we include mobile apps that use 10 or more TPLs and the relevant TPLs in the experiments. The final dataset used in the experiments contains 31,432 Android apps, 752 distinct TPLs and 537,011 app-library usage records.

5.2 Experiment Setup

We employ the cross-validation technique [25] to set up the experiments. First, for each app u in M, we randomly remove $rm \in \{1, 3, 5\}$ libraries to simulate a to-be-improved version of u, i.e., change rm entries from 1 to 0 in the uth row of M. LibSeek is then trained on the remaining app-library usage records in M. The ground truth of the evaluation is that those removed libraries will definitely be useful to the corresponding to-be-improved versions of the apps. Thus, the objective of the evaluation is to find out whether those libraries removed from the apps (referred to as *correct libraries* hereafter) can be predicted by LibSeek. Then, we run LibSeek to make predictions, generating one prediction list $pl(u)$ (referred to as *list* hereafter) for each app $u \in M$ with pn ($pn \geq rm$) libraries. Finally, we inspect the list for the libraries from the corresponding app to evaluate the prediction performance of LibSeek. To simulate various prediction scenarios, we change the values of two parameters, i.e., rm and pn. We also change the values of α and $weight$, which are used in Eq. (12) and Eq. (7), as well as k, i.e., the number of neighbors selected for each app and library. For ease of representation, one round of the validation is referred to as *an experiment instance*. In each experiment instance, the performance of LibSeek is evaluated across all the recommendation lists, one for each app. When the value of a parameter is changed, we run 100 experiment instances and average the results.

All experiments are conducted on a machine with Intel i5-4570 CPU 3.20 GHz, 8 GB RAM, running Windows 7 x 64 Enterprise.

5.3 Performance Metrics

We employ five widely-used metrics to comprehensively evaluate the prediction performance of LibSeek from two perspectives, diversity and accuracy. Those metrics are calculated for each experiment instance, all within the range of [0, 1]. All the metrics indicate better performance when their values are higher. Details of those metrics are as follows.

Coverage and Mean Inter-List Diversity are used the evaluate the diversity in the prediction results of LibSeek.

7. https://mvnrepository.com/
8. https://github.com/

Coverage (COV). Coverage measures the ratio of distinct libraries in all lists to all libraries in Lib in an experiment instance [31]. One COV value is calculated for each experiment instance as follows:

$$COV = \frac{\sum_{u=1}^{m} distinct(pl(u))}{|L|} \tag{25}$$

where $distinct(L)$ is the number of distinct libraries in list L.

A higher COV value indicates that the prediction approach can give more equal opportunities to the libraries in L. If a prediction approach only predicts popular libraries, its COV value will be low.

Mean Inter-List Diversity (MILD). Inter-List Diversity measures the difference between two lists [32] based on their Hamming distance. It is calculated as follows:

$$H_{u,v} = 1 - \frac{pl(u) \bigcap pl(v)}{pn} \tag{26}$$

If $pl(u)$ and $pl(v)$ are the same, their Hamming distance is 0. If they share no libraries at all, their Hamming distance is 1. Thus, a higher $H_{u,v}$ indicates higher diversity across the lists and that the lists are more personalized compared with lists containing only the popular libraries. MILD averages the inter-list diversity between every two lists in an experiment instance.

To evaluate if and how much LibSeek trades accuracy for diversity, Mean Average Precision, Mean Precision and Mean Recall are used to evaluate the prediction accuracy of LibSeek.

Mean Average Precision (MAP). Average precision is widely-used to evaluate the ranking positions of correct libraries in a list. It assigns different weights to the correct libraries at different positions [19] to measures whether LibSeek can put correct libraries at higher positions in the list. Given a list, it is calculated as follows:

$$AP = \frac{\sum_{i=1}^{pn} \frac{p(i)}{i} \times rel(i)}{\sum_{i=1}^{pn} rel(i)} \tag{27}$$

where i is the position of each library in the list, $p(i)$ is the sequence number of the correct library at position i, and $rel(i)$ returns 1 if the ith library in the list is correct and 0 otherwise. For example, given a list with 5 libraries (l_a, l_b, l_c, l_d, l_e), assume that the first and third libraries, i.e., l_a and l_c, are correct libraries. Then the sequence numbers of l_a and l_c are 1 and 2, respectively, i.e., $p(1) = 1$ and $p(3) = 2$. MAP averages the APs across all the lists in one experiment instance.

Mean Precision (MP). The precision of a list $pl(u)$ is the ratio of correct libraries to all libraries in $pl(u)$. MP is the mean precision across all the lists in one experiment instance. Different from MAP, it does not consider the positions of correct libraries in the lists.

Mean Recall (MR). The recall of a list is the ratio of correct libraries in the list to all correct libraries removed from the corresponding app. MR is the mean of the recalls across all the lists in one experiment instance.

5.4 Performance Evaluation

In the experiments, we compare LibSeek with 8 representative approaches:

TABLE 2
Performance Comparison

Attr.	Methods	pn=5					pn=10				
		COV	MILD	MAP	MP	MR	COV	MILD	MAP	MP	MR
rm=1	POP	0.0316	0.5406	0.2840	0.0753	0.3765	0.0465	0.4021	0.2949	0.0457	0.4565
	SGD	0.0319	0.5420	0.2873	0.0761	0.3808	0.0455	0.4017	0.2973	0.0456	0.4556
	ALS	0.0319	0.4959	0.1379	0.0673	0.3364	0.0468	0.4023	0.1550	0.0457	0.4573
	BPR	0.0604	0.5699	0.2911	0.0783	0.3917	0.0867	0.4335	0.3015	0.0469	0.4693
	MPR	0.0348	0.5220	0.0145	0.0074	0.0371	0.0511	0.4005	0.0177	0.0062	0.0615
	CoMF	0.2738	0.8772	0.3748	0.0932	0.4661	0.3210	0.8616	0.3822	0.0521	0.5212
	LibRec	0.2921	0.7932	0.4622	0.1267	0.6335	0.2990	0.6941	0.4669	0.0668	0.6682
	WRMF	0.2973	0.9228	0.5151	0.1337	0.6689	0.3446	0.8939	0.5265	0.0752	0.7528
	LibSeek	**0.3346**	**0.9274**	**0.5236**	**0.1348**	**0.6741**	**0.3960**	**0.8992**	**0.5346**	**0.0755**	**0.7553**
rm=3	POP	0.0322	0.5314	0.5931	0.2147	0.3579	0.0455	0.3913	0.5682	0.1341	0.4468
	SGD	0.0282	0.5289	0.5920	0.2173	0.3621	0.0415	0.3897	0.5694	0.1335	0.4451
	ALS	0.0332	0.5314	0.5686	0.2149	0.3581	0.0455	0.3918	0.5452	0.1342	0.4472
	BPR	0.0864	0.5670	0.5961	0.2250	0.3751	0.1290	0.4337	0.5728	0.1376	0.4586
	MPR	0.0346	0.5118	0.0380	0.0193	0.0322	0.0527	0.3863	0.0491	0.0188	0.0625
	CoMF	0.2731	0.8980	0.5680	0.2279	0.3798	0.3366	0.8619	0.5546	0.1317	0.4392
	LibRec	0.2916	0.8369	0.6883	0.2789	0.4648	0.2936	0.7265	0.6864	0.1542	0.5142
	WRMF	0.2896	0.9013	0.7225	0.3682	0.6137	0.3361	0.8883	0.6930	0.2138	0.7128
	LibSeek	**0.3245**	**0.9135**	**0.7280**	**0.3710**	**0.6183**	**0.3907**	**0.8894**	**0.6971**	**0.2158**	**0.7193**
rm=5	POP	0.0316	0.5065	0.7413	0.3383	0.3383	0.0449	0.3699	0.6813	0.2180	0.4360
	SGD	0.0269	0.5004	0.7376	0.3421	0.3421	0.0402	0.3669	0.6854	0.2160	0.4320
	ALS	0.0319	0.5086	0.7434	0.3361	0.3361	0.0447	0.3699	0.6812	0.2173	0.4347
	BPR	0.1234	0.5420	0.7366	0.3483	0.3483	0.1936	0.4200	0.6808	0.2219	0.4438
	MPR	0.0356	0.4802	0.0595	0.0360	0.0360	0.05423	0.3649	0.0766	0.0320	0.0640
	CoMF	0.2722	0.8978	0.6293	0.3093	0.3093	0.3405	0.8614	0.6052	0.1824	0.3647
	LibRec	0.2885	0.8652	0.6922	0.4400	0.4400	0.2992	0.7619	0.6890	0.2434	0.4868
	WRMF	0.2843	0.9016	0.7816	0.5161	0.5161	0.3301	0.8762	0.7328	0.3206	0.6413
	LibSeek	**0.3141**	**0.9205**	**0.7896**	**0.5291**	**0.5291**	**0.3796**	**0.8810**	**0.7396**	**0.3293**	**0.6587**

1) POP – a popularity-based recommendation approach that always recommends the most popular libraries that have not been used by the current app. This is the baseline approach in the experiments.
2) SGD [20] – a representative MF-based prediction approach that only uses the explicit entries and discards all unknown entries when making predictions.
3) ALS [21] – is another representative MF-based approach. The major difference between ALS and SGD is that they employ different strategies to minimize their loss functions.
4) BPR [22] – is an approach that employs a personalized ranking method to include implicit entries into the prediction. It is optimized for ranking tasks via the maximum posterior estimator derived from a Bayesian analysis.
5) MPR [33] – divides the unobserved items into different parts to further relax the simple pairwise preference assumption into multiple pairwise ranking criteria.
6) CoMF [34] – jointly decomposes the user-item interaction matrix and the item-item co-occurrence matrix with shared item latent factors to get a better precision and coverage.
7) LibRec [19] – this approach combines association rule mining and user-based (app-based) collaborative filtering to recommend libraries for traditional Java projects.
8) WRMF [29] – this approach is the first work to consider both the implicit and the explicit information during the matrix factoring process. Generally, it assigns a same weight to all the explicit entities and assigns 1 to all the implicit entities.

To perform an objective and fair comparison, we tune the parameters employed by each approach to obtain the best results. We set the dimension of latent factors $f = 20$ for all approaches. Specially, we set the maximum number of iterations $iter = 10$ for BPR, SGD, CoMF and LibSeek, 20 for ALS and WRMF and 1000 for MPR. We set the regularization value $\lambda = 0.001$ for SGD and $\lambda = 0.01$ otherwise. We set $weight = 24$, $\alpha = 0.5$, $k = 6$ in LibSeek. Other parameters are specified following the original setup of the experiments in the corresponding papers. We vary rm from 1 to 3 and then to 5, which determines how many libraries are removed from each testing app. We also change pn from 5 to 10, which represents the length of the prediction lists.

5.4.1 Overall Performance Comparison

Table 2 compares the overall performance of the nine approaches with $pn \in \{5, 10\}$ and $rm \in \{1, 3, 5\}$. It shows that LibSeek achieves the best performance in terms of both recommendation diversity and accuracy indicated by all five metrics. On average across six cases with different $rm - pn$ combinations, LibSeek outperforms POP by 836.06%, 102.38%, 37.69%, 64.23%, 64.29%, in terms of COV, MILD, MAP, MP and MR, respectively, SGD by 920.55%, 103.33%, 37.07%, 63.70%, 63.72%, BPR by 256.63%, 85.99%, 36.14%, 59.19%, 59.18%, MPR by 732.99%, 107.34%, 1943.25%, 1334.72%, 1335.69%, respectively. Similar to LibSeek, WRMF makes recommendations based on both explicit and implicit entities. Thus, we can find that LibSeek achieves only slight advantages over WRMF in MAP, MP and MR, i.e., 1.08%, 1.36% and 1.33%, respectively. This comes from LibSeek's adaptive weighting mechanism. Unlike WRMF which assigns a same weight to all the explicit entities, LibSeek assigns different weights to different explicit entities. However, by leveraging the neighborhood information, LibSeek significantly outperforms WRMF in coverage measured by COV, by 13.54% on average. With $rm = 1$ and $pn = 5$, LibSeek achieves the most significant average advantages over the other approaches, i.e., 527.68%, 49.15%, 511.48%, 262.76% and 262.05% in COV, MILD, MAP, MP and MR, respectively. The results presented in Table 2 demonstrate that, when recommending potentially useful TPLs for app developers, **LibSeek offers global diversity by giving opportunities to both popular and less popular TPLs**. **LibSeek also significantly diversifies the TPLs across individual recommendation lists and it does so with high prediction accuracy**. This shows that the recommendations made by LibSeek for individual apps are highly personalized.

Table 2 also shows that, when rm increases from 1 to 5, some of LibSeek's performance advantages over the other approaches decrease. For example, when $pn = 5$ and rm increases from 1 to 5, LibSeek's average performance advantages over the other approaches decrease from 527.68% to 477.23% in COV, 511.48% to 161.83% in MAP, 262.76% to 210.49% in MP and 262.05% to 210.49% in MR. The main reason for the decreases in LibSeek's performance advantages is that more explicit entries in M are discarded when rm increases. As discussed in Section 4.1, LibSeek assigns individualized $weight$ values ($weight > 1$) to explicit entries and a universal $weight$ value ($weight = 1$) to implicit entries. When rm increases, more explicit entries are removed from the testing apps. The removal of these entries impacts LibSeek's recommendation performance more significantly than the other approaches which only treat explicit entries with equal $weight$ values ($weight = 1$) in M. On the other hand, the increase in rm leads to higher advantages of LibSeek in terms of MILD. When $pn = 5$ and rm increases from 1 to 5, LibSeek's average performance advantage over all other approaches increases from 49.15% to 52.40%. When $pn = 10$ and rm increases from 1 to 5, LibSeek's average performance advantage in MILD increases from 79.68% to 85.76%.

5.4.2 Impact of $weight$

As discussed in Section 4.1, the parameter $weight$ is used to set the weight for each entry in M. To evaluate its impact on LibSeek's performance, we vary its value with $rm \in \{1, 3, 5\}$, $pn \in \{5, 10\}$, $k = 6$, $\alpha = 0.5$. According to Eq.(7), when $weight = 0$, LibSeek assigns the same weight, i.e., 1, to all the entries in M and treats both explicit and implicit entries equally. Fig. 5 shows the recommendation results with $pn = 5$ for the first row and $pn = 10$ for the second row.

Compared with $weight = 0$, **LibSeek achieves much higher performance with a non-zero** $weight$. On average across all cases with different $weight > 0$, pn and rm, LibSeek outperforms $weight = 0$ by 34.47%, 1.21%, 20.76%, 30.70% and 30.70% in terms of COV, MILD, MAP, MP and MR, respectively. This indicates that the adaptive weighting mechanism improves the diversity and accuracy of LibSeek's recommendation results by leveraging the explicit entries in M. As $weight$ increases, LibSeek's recommendation performance increases in general in terms of both diversity and accuracy. The metric MILD, which indicates the internal diversity of each prediction list, is an exception. In both Fig. 5(1b) and Fig. 5(2b), when $weight$ increases from 0, LibSeek's MILD values increase gradually. For example, when $rm = 5$ and $weight = 24$, its MILD value increases by 2.62% from 0.9057 to 0.9294 in Fig. 5(1b), and by 3.14% from 0.8678 to 0.8950 in Fig. 5(2b). As $weight$ continues to increase to 44, LibSeek's MILD values start to decrease, by 0.97% from 0.9294 to 0.9205 in Fig. 5(1b) and by 1.56% from 0.8950 to 0.8810 in Fig. 5(2b). However, such slight performance decreases in MILD are acceptable in most, if not all, cases given the performance increase gained in other performance metrics. Let us take a look at Fig. 5(1a) and Fig. 5(2a). When $rm = 5$ and $weight$ increases from 24 to 44, LibSeek's COV values increase by 11.23% from 0.2824 to 0.3141 with $pn = 5$ in Fig. 5(1a) and by 11.09% from 0.3417 to 0.3796 with $pn = 10$ in Fig. 5(2a). In the meantime, LibSeek's MAP, MP and MR values also increase significantly. When $pn = 5$, its MAP value increases by 15.00% from 0.6866 to 0.7896, its MP value by 26.49% from 0.4183 to 0.5291 and its MR value by 26.49% from 0.4183 to 0.5291. When $pn = 10$, its MAP value increases by 14.70% from 0.6448 to 0.7396, its MP value by 22.60% from 0.2686 to 0.3293 and its MR value by 22.62% from 0.5372 to 0.6587. Those increases are much more significant than the slight decreases in the MILD values.

5.4.3 Impact of k

As discussed in Section 4.2, neighborhood information is utilized by LibSeek to increase the diversity in its recommendation results. Parameter k is used to determine how many neighbors of each app or library are utilized by LibSeek in making recommendations. On the one hand, an overly small k may result in loss of information provided by neighbor apps or libraries. On the other hand, an overly large k may introduce dissimilar neighbors to LibSeek and consequently decreases its recommendation performance. To evaluate the impact of k, we change its value from 1 to 10 in the experiments. Fig. 6 presents the results with $rm \in \{1, 3, 5\}$, $\alpha = 0.5$ and $weight = 24$. As demonstrated by Fig. 6(1a), Fig. 6(2a), Fig. 6(1b) and Fig. 6(2b), along with the increase in k from 1 to 10, LibSeek's COV and MILD values increase first, reach their maximums when $k = 6$, then decrease slowly. Take $pn = 5$ as an example. The COV values increase by 5.46% from 0.2873 to 0.3030, by 5.44% from 0.2850 to 0.3005 and 5.27% from 0.2676 to 0.2817 when $rm = 1, 3, 5$, respectively. When k continues to increase, the COV values decrease by 0.73% from 0.3030 to 0.3008, by 0.87% from 0.3005 to 0.2979 and 0.53% from 0.2817 to 0.2802 when $rm = 1, 3, 5$, respectively. On one hand, **it shows that the optimal k can be experimentally obtained for LibSeek to achieve the best performance**. On the other hand, it illustrates that the introduction of neighborhood information increases the global coverage and inner-list diversity in LibSeek's recommendation results. More distinct TPLs can be recommended to developers, for increasing the novelty and serendipity of the recommendation.

More importantly, the introduction of neighborhood information merely affects the recommendation accuracy.

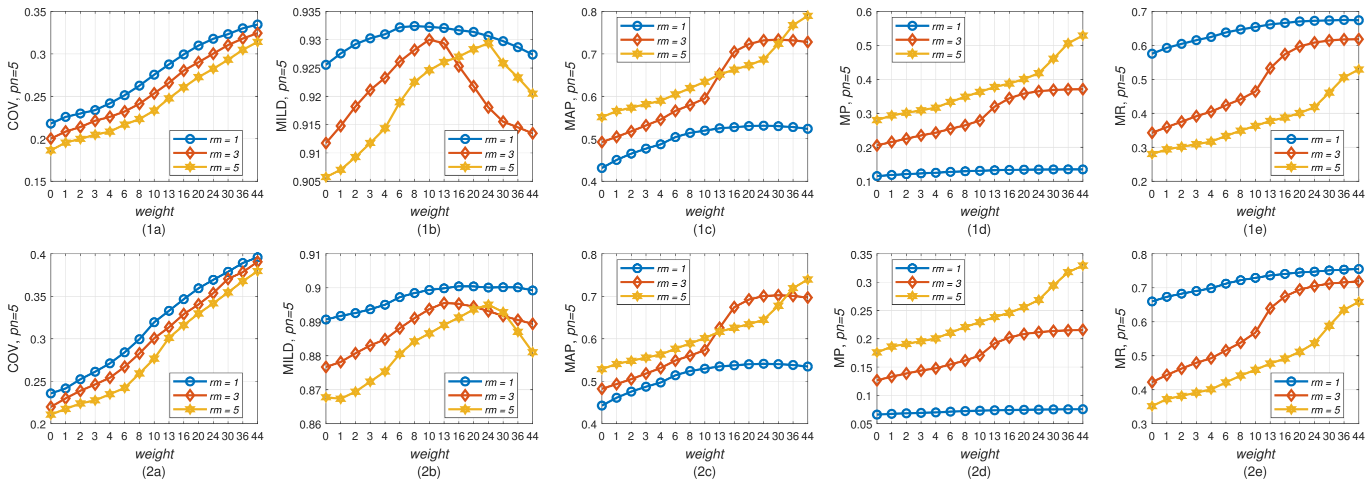


Fig. 5. Impact of $weight$

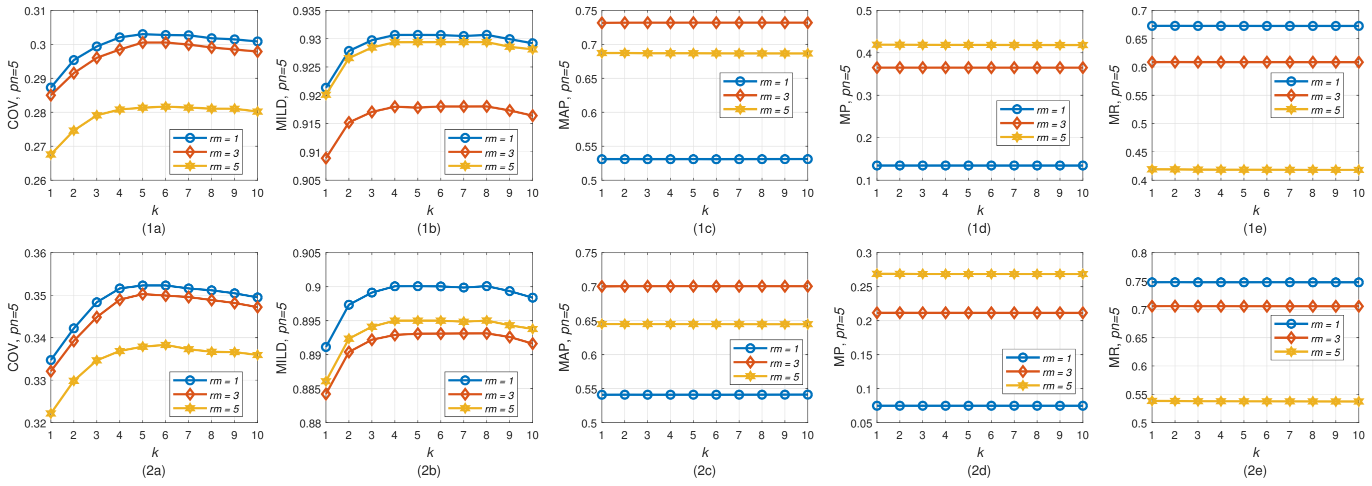


Fig. 6. Impact of k

Looking at the other sub-figures in Fig. 6, we can find that the increase in k does not impact LibSeek's performance in MAP, MP and MR significantly. For example, with $pn = 5$ and $rm = 5$, the maximum impact occurs on MR, with a decrease by only 0.21% from 0.4191 to 0.4182.

5.4.4 Impact of α

Parameter α controls how much LibSeek relies on neighborhood information. To investigate the impact of α, we change its value from 0 to 0.55 in steps of 0.05. The results are shown in Fig. 7 with $rm \in \{1, 3, 5\}$, $pn \in \{5, 10\}$, $k = 6$ and $weight = 24$. When $\alpha = 0$, LibSeek does not leverage any neighborhood information when making recommendations. Thus, we can investigate the effectiveness of the neighborhood information by comparing LibSeek's performance in two different scenarios with $\alpha = 0$ and $\alpha > 0$, respectively. In Fig. 7(1a) and Fig. 7(2a), compared with $\alpha = 0$, LibSeek achieves much higher performance with a non-zero α. On average across all cases with $rm \in \{1, 3, 5\}$, LibSeek's COV outperforms $\alpha = 0$ by 8.66% when $pn = 5$ and by 7.01% when $pn = 10$. When $\alpha = 0.55$, the average COV advantages achieved by LibSeek are 10.15%, 8.15% when $pn = 5$ and $pn = 10$, respectively. This indicates **LibSeek's ability to leverage neighborhood information to improve the coverage of its recommendation results**. As α increases, the increase in α significantly impacts LibSeek's COV values. Take $pn = 5$ as an example. As α increases from 0 to 0.55, LibSeek's COV increases by 9.87% from 0.2755 to 0.3027 when $rm = 1$, by 10.69% from 0.2713 to 0.3003 when $rm = 3$ and by 9.93% from 0.2569 to 0.2824 when $rm = 5$, as shown in Fig. 7(1a). LibSeek's MILD value also increases as long as the α increases, but only slightly. For instance, its MILD value increases by 0.32% from 0.9277 to 0.9307 when $rm = 1$, by 0.34% from 0.9150 to 0.9181 when $rm = 3$ and by 0.44% from 0.9253 to 0.9294 when $rm = 1$, as shown in Fig. 7(1b). Fig. 7(1c) - Fig. 7(1e) and Fig. 7(2c) - Fig. 7(2e) demonstrate phenomena similar to Fig. 6(1c) - Fig. 6(1e) and Fig. 6(2c) - Fig. 6(2e), i.e., the increase in α does not impact LibSeek's recommendation performance significantly. The results presented in Fig. 7 again confirm the conclusion we draw in Section 5.4.3, i.e., the integration of neighborhood information into the prediction model can improve the diversity in LibSeek's prediction results without significantly sacrificing its pre-

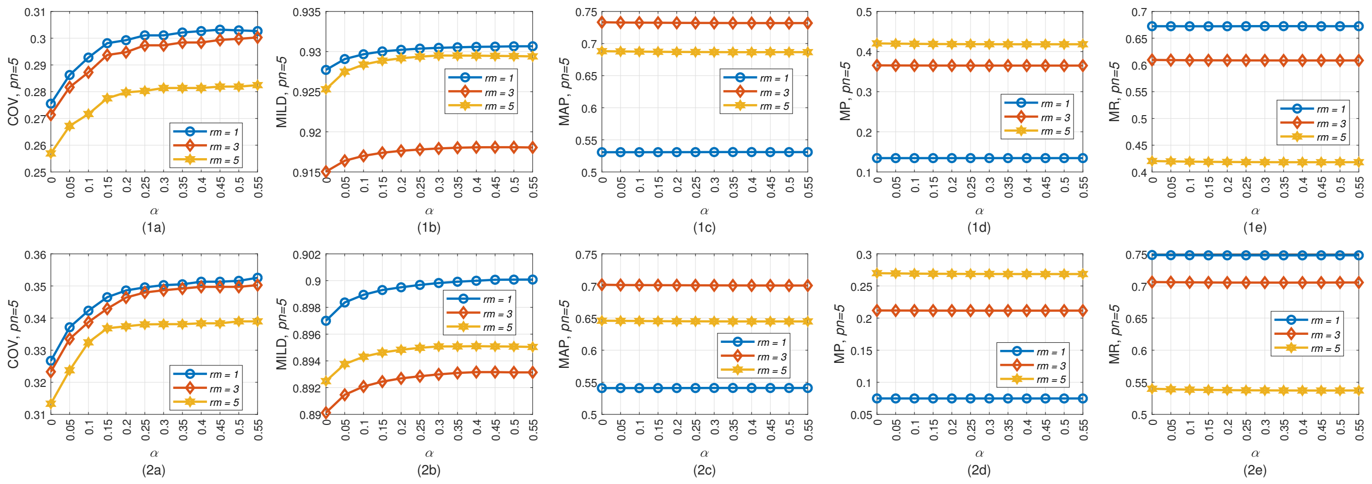


Fig. 7. Impact of α

diction accuracy. This conclusion also can be supported by comparing the results of WRMF with LibSeek. As shown in Table 2, LibSeek has slight advantages over WRMF in MILD, MAP, MP and MR, i.e., on average 0.87%, 1.08%, 1.36%, 1.33% across all rm - pn combinations in terms of MILD, MAP, MP and MR, respectively. However, LibSeek gains a significant advantage over WRMF in COV, by 13.54% on average. Both LibSeek and WRMF assign 1 to the implicit entries in M and higher weights to explicit entries. The advantage of LibSeek over WRMF in COV is achieved by integrating neighborhood information into its predictions.

5.5 User Study

In addition to the above experiments, we also conducted a user study through an email survey to validate LibSeek's effectiveness as well as the impact of the popular bias discussed in Section 2.3. Firstly, we randomly collected 3,400 apps as well as their developers' emails from Google Play. Secondly, we generated two TPL lists for each of the 3,400 apps. The first list was generated via the POP approach discussed in Section 5.4, which always recommends the most popular TPLs that have not been used by the app. The second list was generated by LibSeek. Each list has 10 potential useful TPLs. Thirdly, for each app, we mixed the two lists into one list with redundant TPLs removed, then sent it to the corresponding developers. **In this way, we conducted a single-blind user study** as the developers did not know which TPLs were recommended by the POP approach or LibSeek. We can then analyze their feedback about LibSeek's recommendation results in an objective manner. Developers are required to rate each TPL in the list with one of five optional labels, i.e., $Actioned$, $Actionable$, $Interesting$, $Irrelevant$ and $Misleading$, which stand for five different levels of their interests in the recommended TPLs from high to low. Higher interest in a recommended TPL indicates its greater usefulness for the corresponding app. We stated that we would only collect information that complies with the *Human Research Ethics*[9] and we would provide a $25 Amazon eGift Card for their feedback. In total, we received feedback from 237 developers.

9. https://www.nhmrc.gov.au/

We then calculated and compared the number of developers' different ratings for the TPLs recommended by LibSeek and the POP approach. Fig. 8 shows the distributions of the 4,470 ratings on 474 lists - 237 generated by LibSeek and 237 generated by the POP approach. We can see that developers showed overall positive attitudes towards LibSeek. In total, 75.27% (1,784 out of 2,370) of all the TPLs recommended by LibSeek are rated as $Actionable$ or $Actioned$. In contrast, only 11.73% of the TPLs recommended by the POP approach are rated as $Actionable$ or $Actioned$. This indicates the effectiveness of LibSeek in finding useful TPLs for apps. The survey results show that developers are indeed interested in many TPLs that are potentially useful for their apps despite their lower popularity. This confirms the limitation of recommending only popular TPLs to developers as well as the need for novelty and serendipity in the recommendation results, as discussed in Section 1.

We also analyzed the survey results according to developers' development experience. Fig. 9 shows the distribution of developers' different ratings of the TPLs recommended by LibSeek. We divided developers into four categories according to their development experience levels, i.e., less than 2 years, 2-3 years, 4-5 years and 6-10 years. There are 57, 74, 83, 23 developers in each category, respectively. We can find in Fig. 9 that new developers gave LibSeek' recommendation results higher ratings than experienced developers. Out of all the developers with 2 years' development experience or less, 93.86% of the recommendations made by LibSeek were rated $Actioned$ or $Actionable$. This number is 72.43% for developers with 2-3 years development experience, 67.83% for those with 4-5 years and 65.322% for those with 6-10 years. This tells us that recommending TPLs for developers is useful, especially for relatively new developers.

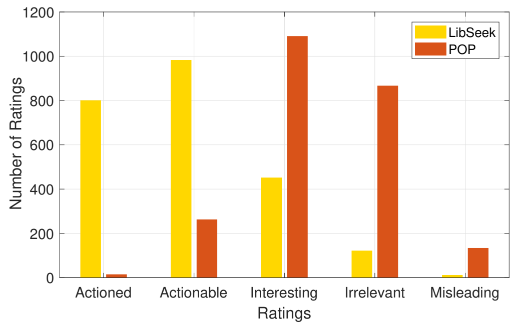


Fig. 8. Distributions of ratings for TPLs recommended by LibSeek and POP

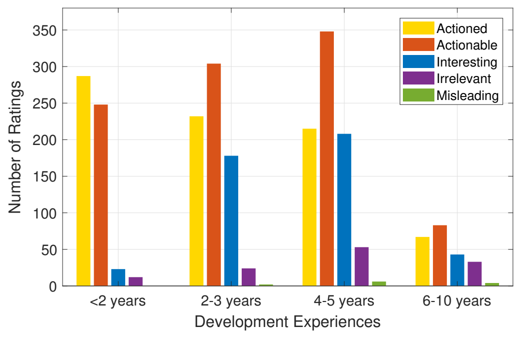


Fig. 9. Distributions of ratings for TPLs recommended by LibSeek by developers' development experience

5.6 Threats to Validity

5.6.1 *Threats to construct validity*

One of the main potential threats to the construct validity of the evaluation is whether the comparison with POP, SGD [20], ALS [21], BPR [22], MBR [33], CoMF [34], WRMF [29] and LibRec [19] can properly demonstrate the recommendation performance of LibSeek. We minimized this threat in four different ways. First, the comparison involves representatives in different categories of approaches, i.e., POP always recommends popular libraries, SGD and ALS consider only the explicit entries in M, BRP and WRMF consider both explicit and implicit entries. Second, we also compared LibSeek with approaches MBR and CoMF that employ extra implicit information, such as multiple pairwise ranking criteria [33] or item-item co-occurrence information [34]. Third, we compared LibSeek with LibRec, which is state-of-the-art approach for predicting libraries for traditional Java projects. Fourth, the approaches are compared in terms of not only the diversity but also the accuracy of their results. Furthermore, we changed five experiment parameters to simulate various recommendation scenarios. In this way, we compare the impacts of those parameters on the approaches comprehensively. The other main threat is the use of the removed explicit entries - the removed TPLs - in M as the ground truth. For each app, useful TPLs are yet to be discovered. When evaluating LibSeek's recommendation accuracy, the incorrect TPLs on a recommendation list are unnecessarily not useful. Thus, the results presented in Section 5.4 indicate the lower bound of LibSeek's prediction accuracy. LibSeek's true prediction accuracy is likely to be higher than what is presented in Table 2 as well as Fig. 5, Fig. 6 and Fig. 7.

5.6.2 *Threats to external validity*

The representativeness of MALib is one of the main potential threats to the external validity of the evaluation. To minimize this threat, we included in MALib a total of 31,432 real-world apps on Google Play, the largest and most popular Android app repository in the world. In addition, both LibSeek and MALib are published online. More apps and TPLs can be included in MALib to further improve its representativeness. Another potential threat is whether our experiments can properly simulate real-world prediction needs. To attack this challenge, we employed cross-validation technique in the experiments. By removing libraries currently used in apps, we mimicked the to-be-improved versions of those apps and used the removed libraries as the ground-truth for the evaluation. This can properly demonstrate LibSeek's ability to fulfill the corresponding app developers' prediction needs. App-library usage records play a significant role in building the ground truth for the experiments. Hence, the accuracy of those records is the third potential threat to the external validity of the evaluation. To minimize this threat, we employed LibRadar [2], a TPL detection tool which was trained over more than 2 million Android apps, to extract the TPLs used by the apps in MALib. We also manually inspected and updated its library dataset against the TPLs on Maven and GitHub to improve the accuracy of LibRadar's detection results.

5.6.3 *Threats to internal validity*

The main threat to the internal validity of the evaluation is about our conclusion of the relationship between LibSeek's high recommendation performance and its diversity and coverage performance. To minimize this threat, we have implemented a popularity-based approach, i.e., POP, as a baseline in the experiments. It allows us to analyze the diversity and coverage of LibSeek's recommendation results via comparison against a popularity-based prediction approach.

5.6.4 *Threats to conclusion validity*

The lack of statistical tests is the main potential threat to the conclusion validity, e.g., we should have conducted chi-square tests to draw conclusions when evaluating LibSeek. However, we ran 100 experiment instances with 31,432 apps per experiment instance and averaged the results each time we varied one of the five experimental parameters. This led to a large number of test cases, which tend to result in a small p-value in the chi-square tests and lower the practical significance of statistical tests [35]. Thus, the threat to the conclusion validity due to the lack of statistical tests might be high but not significant.

6 Related Work

In recent years, more and more effort has been devoted to the detection of third-party libraries used in mobile apps [2] [3] [5] [4] [36] [37] [38]. Early studies usually rely on white-lists, e.g., AdRisk [36] investigates ad plug-ins in apps with a white-list of 100 advertisement libraries. These methods commonly match the package names to find specific libraries and thus are not reliable when apps are obfuscated [3]. Recently, clustering techniques have been employed to detect libraries. Similar modules in each app are clustered using different strategies, such as the hierarchy of packages [37], package relationship [38], the reference and inheritance relationships among classes and methods [3], and the numbers and types of API calls in each folder [2]. In addition, some methods employ feature matching to detect libraries, e.g., LibScout [5] and LibPecker [4]. They compare the similarity between classic libraries and app code modules to detect libraries. In this research, we updated LibRadar's library dataset and used LibRadar to detect TPLs because it is demonstrated to be able to find more libraries than the other approaches [3], [4], [5].

In the field of software engineering, many researchers have employed recommendation techniques to help with mobile app development [39]. Some of them try to exploit the information provided by app users, i.e., user reviews, bug reports, questions, to give a recommendation on development requirements [40] [41] [42] [43]. Some employ software documents or information relevant to developers to recommend libraries [19] or APIs [44]. Thung et al. [19] propose an approach named LibRed that combines association rules and user-based CF to recommend libraries for open-source Java projects. Zheng et al. [44] propose an approach for recommending cross-library APIs through querying Web search engines. Mora and Nadi [45] employ several metrics, e.g., popularity, release frequency, fault-proneness and security to help developers evaluate and compare the quality of libraries. The limitation is that app developers must be aware of what functionality they need while searching for new libraries, which makes it inapplicable for developers to explore these libraries with unknown but potentially useful functionality. LibraryGuru [46] recommends both functional APIs and callback APIs by extracting correlations between functionality descriptions and Android APIs from online tutorials and SDK documents. The major limitation of LibraryGuru is that it can only be used to search for APIs in one library rather than across libraries. In this paper, we attempt to assist app developers in finding potential useful libraries for their apps rather than taking over their responsibilities entirely. Thus, we focus on the prediction of libraries - rather than APIs - that are potentially useful for app development. Furthermore, LibSeek employs existing app-library usage information.

Matrix factorization (MF) and its extensions have gained considerable attention and have become the *de facto* for building modern recommender systems [47]. Many MF-based prediction approaches have been proposed to facilitate or enhance the recommendations for e-commerce websites or news portals. In the early years, SVD [48] was proposed to make recommendations. However, it is not capable of handling sparse user-item matrices effectively and efficiently. More sophisticated MF-based approaches have been proposed to address the issue of data sparsity and improve prediction performance. In general, there are two categories of approaches, one based on SGD [20] and the other based on ALS [21]. However, both categories of approaches consider only explicit information, e.g., numeric ratings. Hu et al. [29] reveal that implicit information can also contribute to making predictions. They propose an approach often referred to as WRMF that assigns confidence to each item according to the amount of the corresponding implicit information. It is implemented and compared in our experiments. Rendel et al. [22] focus on the ranking issues caused by implicit information and propose a generic optimization criterion named BPR for personalized ranking that maximizes the posterior estimator derived from a Bayesian analysis. BRP is also selected as the representative approach that leverages both explicit and implicit information for making predictions in our experimental evaluation. In recent years, several approaches have been proposed to make predictions considering implicit information. The most popular way is to employ external social information, e.g., Ciao,[10] Delicious[11] and LibraryThing,[12] to improve prediction accuracy [49], [50]. However, such external information is not available in TPL recommendation scenarios. Some approaches employ group information to improve the prediction accuracy [33], [34]. To be more specific about the TPL recommendation performance of those approaches, they are implemented in our experiments as MBR and CoMF.

As discussed in Section 2.3, the issue of popularity bias drives a very small number of popular libraries to be included in the prediction results achieved by MF-based approaches. We aim to offer app developers novelty and serendipity by diversifying prediction results. This is achieved through recommending unpopular but useful libraries which may otherwise slip under the radar during app developers' search for useful libraries. Cremonesi et al. [18] find that the very few top popular items can skew the performance of recommender systems and they suggested to choose the testing set carefully to address the accuracy bias. Lathia et al. [32] and Zhang et al. [51] introduce temporal information into MF to improve the diversity of the prediction results. Abdollahpouri [52] introduce a flexible regularization based framework, which categorizes items into a popular set and an unpopular set, to increase the coverage of the prediction results. Unfortunately, in the context of library predictions, the available information does not include timestamps and it is difficult to manually and accurately categorize libraries into popular and unpopular ones. Thus, LibSeek attempts to address the issue of popularity bias with a new personalized weighting mechanism that assigns a personalized weight to each library according to its popularity.

MF-based approaches employ only global information in the user-item matrix to make predictions. This is a major reason for the lack of diversity in their prediction results. Memory-based prediction approaches based on users [12], items [13] or both [14] leverage neighborhood information to make predictions, i.e., information about similar items and similar users. Inspired by memory-based prediction approaches, LibSeek employs both apps' and libraries' information neighborhood to diversify the prediction results. The experimental results presented and discussed in Section 5 demonstrate that LibSeek outperforms classic approaches in terms of diversity as well as accuracy.

10. https://www.ciao.co.uk
11. https://www.delicious.com
12. https://www.librarything.com

7 Conclusion and Future Work

In this paper, we first revealed the importance of third-party libraries (TPLs) in mobile app development with the results of our investigation of 31,432 apps on Google Play. Then, also based on the results of our investigation, we implemented three representative model-based MF approaches to recommend useful third-party libraries for those apps. However, the results showed that existing approaches suffer from low diversity due to popularity bias. To address this issue, we proposed LibSeek, a novel approach for recommending potentially useful TPLs for mobile apps. To tackle the issue of popularity bias, LibSeek employs a personalized weighting mechanism and neighborhood information to diversify the recommendation results. Extensive experiments show that LibSeek significantly outperforms all the eight representative recommendation approaches in terms of both the diversity and accuracy of the recommendation results.

As discussed in Section 5.6.1, the prediction performance of LibSeek is reliant on the quality of the data in the MALib dataset. In our future work, we will extend MALib further with more mobile apps and more TPLs. We will also try to find new and more accurate TPL detection tools for improving the accuracy of the information in MALib. In addition, we will try to leverage additional information about apps and TPLs, e.g., their categories, versions, etc., to further improve LibSeek's performance.

Acknowledgement

This work is partly funded by Australian Research Council Discovery Projects DP170101932, DP180100212 and Laureate Fellowship FL190100035. The corresponding author of this paper is Yun Yang.

References

[1] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2187–2200.

[2] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 653–656.

[3] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 335–346.

[4] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 141–152.

[5] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 356–367.

[6] X. Su and T. M. Khoshgoftaar, "A survey of collaborative filtering techniques," *Advances in artificial intelligence*, vol. 2009, 2009.

[7] Y. Shi, M. Larson, and A. Hanjalic, "Collaborative filtering beyond the user-item matrix: a survey of the state of the art and future challenges," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 3:1–3:45, May 2014.

[8] Z. Zheng and M. R. Lyu, "Personalized reliability prediction of web services," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 2, p. 12, 2013.

[9] Y. Zhang, K. Wang, Q. He, F. Chen, S. Deng, Z. Zheng, and Y. Yang, "Covering-based web service quality prediction via neighborhood-aware matrix factorization," *IEEE Transactions on Services Computing*, pp. 1–1, 2019.

[10] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins, "Eigentaste: A constant time collaborative filtering algorithm," *Information Retrieval*, vol. 4, no. 2, pp. 133–151, 2001.

[11] M. D. Ekstrand, J. T. Riedl, and J. A. Konstan, "Collaborative filtering recommender systems," *Foundations and Trends in Human-Computer Interaction*, vol. 4, no. 2, pp. 81–173, 2011.

[12] J. Herlocker, J. Konstan, A. Borchers, and J. Riedl, "An algorithmic framework for performing collaborative filtering," in *22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 1999, pp. 230–237.

[13] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *10th International Conference on World Wide Web*. ACM, 2001, pp. 285–295.

[14] J. Wang, A. P. de Vries, and M. J. T. Reinders, "Unifying user-based and item-based collaborative filtering approaches by similarity fusion," in *29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2006, pp. 501–508.

[15] G.-R. Xue, C. Lin, Q. Yang, W. Xi, H.-J. Zeng, Y. Yu, and Z. Chen, "Scalable collaborative filtering using cluster-based smoothing," in *28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2005, pp. 114–121.

[16] T. Hofmann, "Latent semantic models for collaborative filtering," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 89–115, 2004.

[17] R. Bell, Y. Koren, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, pp. 30–37, 08 2009.

[18] P. Cremonesi, Y. Koren, and R. Turrin, "Performance of recommender algorithms on top-n recommendation tasks," in *4th ACM Conference on Recommender Systems*. ACM, 2010, pp. 39–46.

[19] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 182–191.

[20] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *17th International Conference on Knowledge Discovery and Data Mining*. ACM, 2011, pp. 69–77.

[21] Y. Koren, "Factorization meets the neighborhood: A multifaceted collaborative filtering model," in *14th International Conference on Knowledge Discovery and Data Mining*. ACM, 2008, pp. 426–434.

[22] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "Bpr: Bayesian personalized ranking from implicit feedback," in *25th Conference on Uncertainty in Artificial Intelligence*, 2009, pp. 452–461.

[23] Y.-J. Park and A. Tuzhilin, "The long tail of recommender systems and how to leverage it," in *2008 ACM Conference on Recommender Systems*. ACM, 2008, pp. 11–18.

[24] M. Kaminskas and D. Bridge, "Diversity, serendipity, novelty, and coverage: a survey and empirical analysis of beyond-accuracy objectives in recommender systems," *ACM Transactions on Interactive Intelligent Systems*, vol. 7, no. 1, p. 2, 2017.

[25] S. Arlot, A. Celisse *et al.*, "A survey of cross-validation procedures for model selection," *Statistics surveys*, vol. 4, pp. 40–79, 2010.

[26] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[27] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[28] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *33rd International Conference on Software Engineering*. IEEE, 2011, pp. 481–490.

[29] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *8th IEEE International Conference on Data Mining*. IEEE, 2008, pp. 263–272.

[30] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.

[31] M. Ge, C. Delgado-Battenfeld, and D. Jannach, "Beyond accuracy: Evaluating recommender systems by coverage and serendipity," in *4th ACM Conference on Recommender Systems*. ACM, 2010, pp. 257–260.

[32] N. Lathia, S. Hailes, L. Capra, and X. Amatriain, "Temporal diversity in recommender systems," in *33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2010, pp. 210–217.

[33] R. Yu, Y. Zhang, Y. Ye, L. Wu, C. Wang, Q. Liu, and E. Chen, "Multiple pairwise ranking with implicit feedback," in *27th ACM International Conference on Information and Knowledge Management*. ACM, 2018, pp. 1727–1730.

[34] D. Liang, J. Altosaar, L. Charlin, and D. M. Blei, "Factorization meets the item embedding: regularizing matrix factorization with item co-occurrence," in *10th ACM Conference on Recommender Systems*. ACM, 2016, pp. 59–66.

[35] M. Lin, H. C. Lucas Jr, and G. Shmueli, "Too big to fail: large samples and the p-value problem," *Information Systems Research*, vol. 24, no. 4, pp. 906–917, 2013.

[36] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 101–112.

[37] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of android ad libraries using semantic analysis," in *IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, 2014, pp. 1–6.

[38] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 89–103.

[39] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[40] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 106–117.

[41] L. V. Galvis Carreño and K. Winbladh, "Analysis of user comments: An approach for software requirements evolution," in *2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 582–591.

[42] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh, "Why people hate your app: Making sense of user feedback in a mobile app store," in *the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2013, pp. 1276–1284.

[43] X. Gu and S. Kim, ""what parts of your apps are loved by users?" (t)," in *30th International Conference on Automated Software Engineering*, 2015, pp. 760–770.

[44] W. Zheng, Q. Zhang, and M. Lyu, "Cross-library API recommendation using web search engines," in *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 480–483.

[45] F. L. de la Mora and S. Nadi, "Which library should i use?: A metric-based comparison of software libraries," in *40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 2018, pp. 37–40.

[46] W. Yuan, H. H. Nguyen, L. Jiang, and Y. Chen, "Libraryguru: Api recommendation for android developers," in *40th International Conference on Software Engineering: Companion*. ACM, 2018, pp. 364–365.

[47] D. Liang, J. Altosaar, L. Charlin, and D. M. Blei, "Factorization meets the item embedding: Regularizing matrix factorization with item co-occurrence," in *10th ACM Conference on Recommender Systems*. ACM, 2016, pp. 59–66.

[48] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numerische Mathematik*, vol. 14, no. 5, pp. 403–420, Apr 1970.

[49] G. Guo, J. Zhang, F. Zhu, and X. Wang, "Factored similarity models with social trust for top-n item recommendation," *Knowledge-Based Systems*, vol. 122, pp. 17 – 25, 2017.

[50] T. Zhao, J. McAuley, and I. King, "Leveraging social connections to improve personalized ranking for collaborative filtering," in *23rd International Conference on Information and Knowledge Management*. ACM, 2014, pp. 261–270.

[51] X. Zhang, J. Zhao, and J. C. Lui, "Modeling the assimilation-contrast effects in online product rating systems: Debiasing and recommendations," in *11th ACM Conference on Recommender Systems*. ACM, 2017, pp. 98–106.

[52] H. Abdollahpouri, R. Burke, and B. Mobasher, "Controlling popularity bias in learning-to-rank recommendation," in *11th ACM Conference on Recommender Systems*. ACM, 2017, pp. 42–46.

Qiang He received his first PhD degree from Swinburne University of Technology, Australia, in 2009 and his second PhD degree in computer science and engineering from Huazhong University of Science and Technology, China, in 2010. He is a senior lecturer at Swinburne. His research interests include software engineering, edge computing, service computing and cloud computing. More details about his research can be found at https://sites.google.com/site/heqiang/.

Bo Li received the BS and MS degree from the School of Information Science and Technology from Shandong Normal University in 2003 and 2010, respectively. He worked as an academic visitor at Shandong University from 2014 to 2015 and at Swinburne University of Technology in 2018. He is currently a Ph.D. student at Swinburne. His research interests include software engineering, edge computing and cybersecurity.

Feifei Chen received her PhD degree from Swinburne University of Technology, Australia in 2015. She is a lecturer at Deakin University. Her research interests include software engineering, cloud computing and green computing.

John C. Grundy received the BSc (Hons), MSc, and PhD degrees in computer science from the University of Auckland, New Zealand. He is currently a professor of software engineering at Monash University, Melbourne, Australia. He is an associate editor of the IEEE Transactions on Software Engineering, the Automated Software Engineering Journal, and IEEE Software. His current interests include domain-specific visual languages, model-driven engineering, large-scale systems engineering, and software engineering education. More details about his research can be found at https://sites.google.com/site/johncgrundy/.

Xin Xia is an ARC DECRA Fellow and a lecturer at the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. Xin received both of his Ph.D and bachelor degrees in computer science and software engineering from Zhejiang University in 2014 and 2009, respectively. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. More information at: https://xin-xia.github.io/.

Yun Yang received his PhD degree from the University of Queensland, Australia, in 1992, in computer science. He is currently a full professor in the School of Software and Electrical Engineering at Swinburne University of Technology, Melbourne, Australia. His research interests include software technologies, cloud computing, workflow systems, and service computing.