Embedding App-Library Graph for Neural Third Party Library Recommendation

Bo Li
Swinburne University of Technology
Melbourne, Australia
boli@swin.edu.au

Qiang He
Swinburne University of Technology
Melbourne, Australia
qhe@swin.edu.au

Feifei Chen
Deakin University
Melbourne, Australia
feifei.chen@deakin.edu.au

Xin Xia
Monash University
Melbourne, Australia
xin.xia@acm.org

Li Li
Monash University
Melbourne, Australia
Li.Li@monash.edu

John Grundy
Monash University
Melbourne, Australia
john.grundy@monash.edu

Yun Yang
Swinburne University of Technology
Melbourne, Australia
yyang@swin.edu.au

ABSTRACT

The mobile app marketplace has fierce competition for mobile app developers, who need to develop and update their apps as soon as possible to gain first mover advantage. Third-party libraries (TPLs) offer developers an easier way to enhance their apps with new features. However, how to find suitable candidates among the high number and fast-changing TPLs is a challenging problem. TPL recommendation is a promising solution, but unfortunately existing approaches suffer from low accuracy in recommendation results. To tackle this challenge, we propose GRec, a graph neural network (GNN) based approach, for recommending potentially useful TPLs for app development. GRec models mobile apps, TPLs, and their interactions into an app-library graph. It then distills app-library interaction information from the app-library graph to make more accurate TPL recommendations. To evaluate GRec's performance, we conduct comprehensive experiments based on a large-scale real-world Android app dataset containing 31,432 Android apps, 752 distinct TPLs, and 537,011 app-library usage records. Our experimental results illustrate that GRec can significantly increase the prediction accuracy and diversify the prediction results compared with state-of-the-art methods. A user study performed with app developers also confirms GRec's usefulness for real-world mobile app development.

CCS CONCEPTS

• **Software and its engineering** → **Application specific development environments**.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468552

KEYWORDS

third-party library, recommendation, mobile app development, app-library graph, graph neural network

ACM Reference Format:
Bo Li, Qiang He, Feifei Chen, Xin Xia, Li Li, John Grundy, and Yun Yang. 2021. Embedding App-Library Graph for Neural Third Party Library Recommendation. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468552

1 INTRODUCTION

Mobile app development has been growing tremendously fast. For example, there are over 2.56 million mobile apps available on the Google Play in the first quarter of 2020[1]. This leads to a very fierce market competition between app vendors/developers. To gain a first mover advantage in such a competitive market, mobile app developers need to release their new apps online as soon as possible. They then need to keep updating their apps to accommodate users' rapidly evolving needs and quickly respond to user reviews. To reduce developers' implementation efforts, a large number of third-party libraries (TPLs) have been published and made available that offer a variety of features [26, 29]. Leveraging TPLs is an effective way to accelerate mobile apps' development and update cycles.

TPLs are playing a more and more significant role in the mobile app ecosystem [1, 18, 45]. Instead of developing everything from scratch, using TPLs can help developers accelerate development, deliver new features, and enhance the overall software quality [26]. For example, compared with newly developed code, many bugs and deficiencies may have already been discovered and fixed in TPLs offering similar or the same functionalities [23]. Recent studies have found that developers regularly attempt to find and use TPLs for their mobile apps [7]. This is also evidenced by a large-scale study revealing that Android apps on Google Play are using 11.81 TPLs on average [9] and on average 60% of the source code of a mobile

[1] https://www.statista.com/statistics/289418/

app comes from TPLs in use [22]. The popularity and advantages of TPLs have led to the development and publication of a large number of TPLs for mobile apps. For example, a total of 6,200 TPLs is hosted on Android Arsenal[2], a popular Android developer portal.

However, this trend means that mobile app developers are now facing significant challenges when seeking useful TPLs for their mobile apps [17, 30, 39, 43]. First, given the severe time-to-market constraints and tremendous available TPLs, it is infeasible for them to manually inspect a great many TPLs to evaluate their usefulness from different perspectives, e.g., functionality, performance, interface, etc. [15]. Second, TPLs are often used by mobile apps in combinations to collectively perform specific functions [26, 28]. Finding appropriate TPL combinations is another time-consuming process for developers [23, 39]. Besides, the rapid evolution of TPLs also contributes to the sophistication in mobile app developers' search for useful TPLs [7]. Thus, developers are in urgent need of help with finding useful TPLs effectively and efficiently.

Collaborative filtering (CF) has been widely and successfully applied in a variety of domains for its ability to mine latent patterns without contextual information [12]. It has been employed to recommend libraries for developers [9, 23, 33]. LibRec combines CF and association rule mining to recommend TPLs for Java projects [33]. It is one of the earliest attempts at CF-based library recommendations. Nguyen et al. proposed CrossRec to recommend TPLs for open-source software projects [23]. Similar to LibRec, CrossRec exploits project similarity when making recommendations. To improve recommendation performance, it adjusts the weight of each TPL based on term frequency–inverse document frequency (TF-IDF). Very recently, LibSeek was proposed for recommending TPLs specifically for mobile apps [9]. It makes recommendations based on matrix factorization and employs an adaptive weighting mechanism to diversify recommendation results.

However, solely based on a two-dimensional app-library matrix, CF-based TPL recommendation approaches exploit only a small portion of low-order app-library interaction information, i.e., app-library usage information, to make recommendations. For example, given a target mobile app A, LibRec [33], and CrossRec [23] find A's top k most similar apps and/or similar TPLs and recommend TPLs that have been used by those apps but not by A. These similar apps and TPLs are identified based on their similarity to A in the use of TPLs. In addition, some useful high-order app-library interaction information that can further improve TPL recommendation accuracy is overlooked. For example, assume that app B is similar to A and app C is similar to B, the app-library interaction information contained in C can also contribute to TPL recommendations for A. However, existing CF-based approaches do not exploit such information and thus suffer from low recommendation accuracy. This is an inherent limitation of existing CF-based approaches.

We tackle the TPL recommendation problem from a new perspective. We model mobile apps, TPLs, and app-library interactions as an *app-library graph*, where mobile apps and TPLs are modelled as nodes[3] and app-library interactions as edges. Apps' low-order app-library interaction information is represented by the links to their 1-hop neighbor library nodes in the graph. At the same time, high-order app-library interaction information can also be extracted from the graph for TPL recommendations. Take apps A, B, and C above for example. Over the graph, there will be one or many paths between C and A through B. Along such paths, more similar apps and TPLs can be identified and high-order app-library interaction information can be extracted from nodes multiple hops away from A, such as C, over the graph to make TPL recommendations for A.

To leverage both low-order and high-order app-library interaction information, we employ a graph neural network (GNN) to make recommendations based on the app-library graph. This is motivated by the fact that GNN is well known for its ability to mine both low-order and high-order information from graphs. It can capture information for a target node from its neighbor nodes within multiple hops [41]. Compared with a baseline and three state-of-the-art approaches, our approach GRec makes more accurate and diversified TPL recommendations. Key contributions of this research includes:

- We make the first attempt to model app-library interactions as a graph. This allows more information among apps and TPLs to be captured.
- Using this app-library graph, we propose a graph neural network based approach, namely GRec, to recommend potentially useful TPLs for mobile apps.
- We train GRec on a large-scale public dataset[4] that contains 61,722 Android apps, 827 distinct TPLs, and 725,502 app-library usage records.
- We conduct extensive experiments and a user study to evaluate the performance of GRec for recommendation accuracy and diversity. The prototype of GRec is published for the validation and reproduction of our experimental results[5].

The remainder of this paper is organized as follows. Section 2 motivates our study. Section 3 introduces the methodology of GRec. Section 4 reports the results of experiments conducted on GRec. Section 5 reviews related work. Finally, Section 6 concludes this paper and points out future work.

2 MOTIVATING EXAMPLE

Fig. 1 demonstrates an example app-library graph (denoted as G) generated based on the interactions among five mobile apps, denoted as $A_1, ..., A_5$, and seven TPLs, denoted as $L_1, ..., L_7$. An edge between an app and a TPL corresponds to the use of the TPL in the app. For example, the edge between A_1 and L_1 indicates that A_1 uses L_1.

Using this app-library graph G, given a target node in G, its low-order app-library interaction information can be obtained from all the nodes that connect to it via one hop in G. Take mobile app A_1 in Fig. 2 as an example. A_1 uses three TPLs, i.e., L_1, L_2, and L_3, because they are directly connected to A_1 in G.

Besides low-order app-library interaction information, high-order app-library interaction information can also be obtained from graph G. For example, in Fig. 2 nodes A_2, A_3, and A_4 are connected to A_1 via 2 hops measured by the number of edges between them. These nodes also contain rich information that might be leveraged for TPL recommendation. For example, path A_2-L_1-A_1 indicates

[2] https://android-arsenal.com/

[3] In this paper, we speak of apps, TPLs, and the corresponding nodes interchangeably.

[4] https://github.com/malibdata/MALib-Dataset, proposed in [9]

[5] The source code of GRec is available at https://github.com/fio1982/GRec.

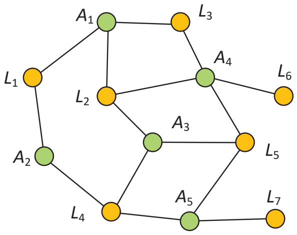

Figure 1: Example app-library graph G

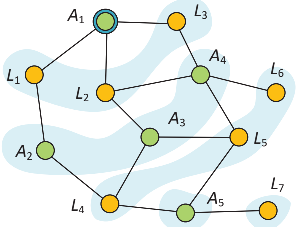

Figure 2: Information useful for A_1 possessed in G

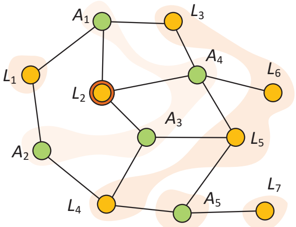

Figure 3: Information useful for L_1 possessed in G

that A_2 has *behaviors* similar to A_1 as both of them connected to L_1 in G. State-of-the-art CF-based library recommendation approaches like LibRec [33] and CrossRec [23] find similar mobile apps for A_1 based on such behaviors. Usually, they select a small number of such similar apps for making recommendations for a target app. This means they inevitably fail to leverage the information provided by all the apps connected to the target app via 2 hops in G and some useful information may be overlooked.

When we take a look at apps further away from the target apps in G, more information can be obtained. Nodes L_4, L_5, and L_6 are connected to A_1 via 3 hops. The information provided by L_4, L_5, and L_6 may also be useful for making TPL recommendations for A_1. For example, from path L_4-A_2-L_1-A_1, we can find that L_1 and L_4 are both used by A_2. There might be a specific correlation between them. For example, A_2 as well as other apps may use them together to perform a specific function, like the use of *Facebook* library and *Picasso* library in combination by a lot of apps reported in [9]. Thus, an app, such as A_1, that solely uses L_1 might find L_4 useful. When we look further away from A_1, we can see that A_5 is not similar to A_1 in terms of TPL usage. However, A_3 and A_5 are both connected to L_4 and L_5 because they both use L_4 and L_5. This indicates that A_3 and A_5 share common interests in L_4 and L_5. For example, they may provide similar or the same functionalities or features through the use of L_4 and L_5. Similarly, A_1 and A_3 share common interests in L_2. Although A_1 and A_5 do not use any common TPLs, A_5 may still share common interests with A_1 as they both share common interests with A_3. Thus, A_5 can also contribute useful information to making TPL recommendations for A_1, then L_7 may be recommended for A_1. However, such high-order information is overlooked by CF-based approaches.

Different nodes with the same distance from a target node may have different impacts on TPL recommendations for the target node. For example, in Fig. 2 three node A_2, A_3, and A_4 have the same distance from A_1 (marked with the same color in Fig. 2) - they are all connected to A_1 via 2 hops in G. However, they do not contribute equally to the TPL recommendations for A_1. For example, A_2 and A_3 are both connected to A_1 via one path each, i.e., A_2-L_1-A_1 and A_3-L_2-L_1, respectively. However, A_4 is connected to A_1 via two paths, i.e., A_4-L_2-A_1 and A_4-L_3-A_1. This indicates that A_4 shares two common TPLs with A_1 while A_2 and A_3 share only one TPL with A_1 each. Thus, A_4 is more similar to A_1 than A_2 and A_3 in their use of TPLs and can contribute more information to TPL recommendations for A_1 than A_2 and A_3.

The above also applies to finding similar TPLs for making TPL recommendations. Take library node L_2 (used by A_1) in Fig. 3 as an example. We can see that L_2 and L_3 are connected to both A_1 and A_4, i.e., they are used by both A_1 and A_4. Thus, L_3 is similar to L_2 in terms of TPL usage. Similarly, L_5 is similar to L_2 as they are both connected to A_3 and A_4. Such low-order information can be exploited by CF-based recommendation approaches. When we look further away from L_2, we can see that L_4 and L_5 are both connected to A_3 and A_5. Thus, L_4 is similar to L_5. Although L_4 is not similar to L_2 in terms of TPL usage, L_4 is similar to L_2 as they are both similar to L_5. Thus, L_4 can contribute high-order information to TPL recommendations for A_1.

When mining the two-dimensional app-library matrix, **CF-based approaches can only leverage low-order app-library interaction information, i.e., the direct interactions between apps and TPLs, to make TPL recommendations**. A new approach that can also leverage the high-order app-library interaction information possessed in the app-library graph is needed to improve recommendation accuracy and save developers' time in finding useful TPLs.

3 GREC APPROACH

Inspired by the great success of neural network based recommendation approaches in various domains [11, 44], GRec is our innovative GNN-based approach for TPL recommendation. GRec employs a graph neural network to make TPL recommendations through distilling both low-order and high-order app-library interaction information from the app-library graph.

3.1 Process Overview

To recommend potentially useful TPLs for a target mobile app, assume A_1, the general process of GRec consists of four phases, as shown in Fig. 4. Its input is the app-library graph G built automatically by GRec based on A_1's library usage records and existing app-library usage records in the training set. Those app-library usage records can either be gathered from developers' input[6] or be extracted with existing tools like LibPecker [45], LibScout [1], LibD [18], and LibRadar [22], similar to [9]. In **Phase 1 (Representation)**, GRec creates an individual latent factor vector for each node in G, including the app nodes and the library nodes. The representations of those nodes are used as the input to Phase 2. In **Phase 2 (Information Distillation)**, GRec employs the GNN to distill both low-order and high-order app-library interaction information for each node in G. This GNN has multiple layers. The first layer is

[6] For example, a developer can provide a list of TPLs used (or to be used) by their mobile apps as the input to GRec.

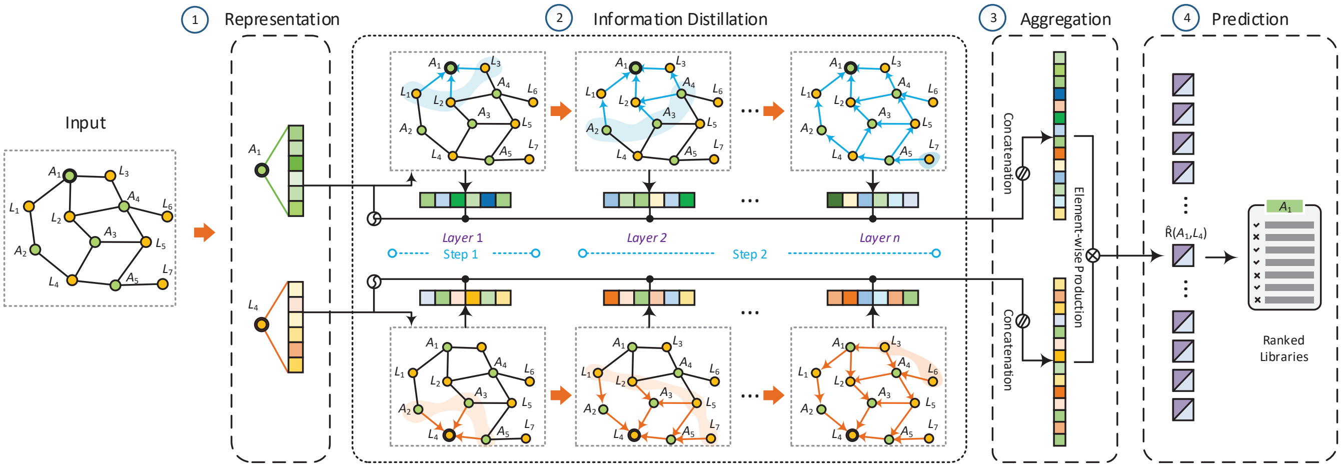

Figure 4: General process of GRec

used to distill low-order app-library interaction information. The other layers are used to distill high-order app-library interaction information. Each layer distills information from neighbor nodes with the same hops. For example, the second layer and third layer distill information for each node from its 2-hop and 3-hop neighbor nodes, respectively. Then, the captured information is aggregated in **Phase 3 (Aggregation)**. At the end of Phase 3, a new vector is generated for each node to accommodate the information captured. Finally in **Phase 4 (Prediction)**, GRec recommends the top nr most useful TPLs that are not used by A_1.

Usage Example: *Bob wants to explore new TPLs for enhancing his app. Without GRec, he has to manually find and inspect a large number of TPLs on Maven, take a long time to read their documents and test their functionalities. With GRec, Bobs can specify a list of TPLs currently used in his app as input. GRec will recommend a number of TPLs potentially useful for his app, say 10. Bob can inspect these TPLs with priority and see if they are useful. In summary, GRec makes recommendations rather than decisions for developers.*

Please note that GRec can make recommendations for mobile apps both published or still under development. A mobile developer can also just provide a few TPLs that they prefer to use in their mobile app as the input for GRec to make TPL recommendations. During the development of a mobile app, GRec can be used to iteratively explore potentially useful TPLs. When the app-library graph used by GRec is updated, GRec can be easily and efficiently retrained to improve its recommendation accuracy. This also allows GRec to include emerging TPLs in its recommendations.

3.2 Phase 1: Representation

Similar to recommendation approaches based on matrix factorization [4] and neural networks [11, 36], GRec embeds both mobile apps and TPLs in a d-dimensional latent factor space, where each mobile app A_i is represented by a latent factor vector $\overrightarrow{A_i} \in \mathbb{R}^d$, and each TPL L_j by a latent factor vector $\overrightarrow{L_j} \in \mathbb{R}^d$, both with random initial values. APPs (and TPLs) are mapped to this space where similar apps (and TPLs) in terms of their features, such as functionality, interface, performance, compatibility, reliability, popularity, dependency, security, are close to each other. The usefulness of a TPL for an app can be evaluated by their distance in this space measured based on their latent features. Accordingly, vector $\overrightarrow{L_j}$ indicates the degree of TPL L_j's possession of each of these latent features. Vector $\overrightarrow{A_i}$ models the degrees of mobile app A_i's preferences for these latent features. GRec is able to approximate the potential usefulness of a TPL L_j for a mobile app A_i in terms of those latent features by performing the element-wise product operation over their vectors as follows:

$$\hat{R}_{i,j} = \overrightarrow{A_i} \cdot \overrightarrow{L_j} \tag{1}$$

where symbol $(\cdot)$ denotes the inner product of two vectors that measures the distance between A_i and L_j in the latent factor space.

3.3 Phase 2: Information Distillation

In this phase, GRec distills information for each node in G from the latent factor vectors of its neighbor nodes. It goes through two main steps: 1) distilling low-order app-library interaction information, 2) distilling high-order app-library interaction information.

Let $\overrightarrow{A_i}_{(0)}$ denote the latent factor vector for mobile app A_i after it is initialized in Phase 1 (Representation). It becomes $\overrightarrow{A_i}_{(1)}$ after it is updated by the first layer of the GNN and $\overrightarrow{A_i}_{(l)}$ after the l-th layer. Similarly, $\overrightarrow{L_j}_{(0)}$ denotes the latent factor vector for library node L_j initialized in Phase 1 (Representation), and $\overrightarrow{L_j}_{(l)}$ after the l-th layer in the GNN. In G, each app node A_i is directly connected to a number of library nodes, denoted as $N(A_i)$. Similarly, the set of app nodes directly connected to a library node L_j is denoted as $N(L_j)$.

3.3.1 *Step 1: Low-order app-library interaction information distillation.*

In this step, GRec employs the first layer of GNN to distill the low-order app-library interaction information for each node from its 1-hop neighbors over G, i.e., $N(A_i)$ for app node A_i and $N(L_j)$ for library node L_j. Take app A_1 and library L_4 in Fig. 4 as example. For mobile app A_1, GRec distills the low-order app-library interaction information from nodes L_1, L_2, and L_3. For TPL L_4, GRec distills the low-order app-library interaction information from nodes A_2, A_3 and A_5. Given an app A_i and a library $L_j \in N(A_i)$, $S_{(1)}(L_j, A_i)$

denotes the low-order app-library interaction information distilled from L_j for A_i, calculated as follows:

$$S_{(1)}(L_j, A_i) = W_1^{(1)}\overrightarrow{L_j}_{(0)} + W_2^{(1)}(\overrightarrow{A_i}_{(0)} \odot \overrightarrow{L_j}_{(0)}) \tag{2}$$

where $W_1^{(1)}, W_2^{(1)} \in \mathbb{R}^{d*d}$ are two weight matrices obtained through the GNN training process. They collectively determine how much information will be extracted from neighbor node L_j. The superscript (1) indicates that $W_1^{(1)}, W_2^{(1)}$ belong to the first layer of the GNN. The symbol $\odot$ denotes the element-wise product operation. The $\overrightarrow{A_i}_{(0)} \odot \overrightarrow{L_j}_{(0)}$ part in (2) includes extra information distilled from libraries that possess latent features preferred by A_i. For example, $\overrightarrow{A_1}_{(0)}$=<0.8,0.2,0.9> means that app A_i prefers the first and the third latent features more than the second one. Let us assume that library L_1's latent factor vector for the first layer of the GNN is $\overrightarrow{L_1}_{(0)}$=<0.7,0.9,0.2>. This indicates that L_1 possesses the first and second latent features more than the third one. With the $\odot$ operation, there is $\overrightarrow{A_1}_{(0)} \odot \overrightarrow{L_1}_{(0)}$=<0.56,0.18,0.18>. It will include additional information in the calculation of $S_{(1)}(L_1, A_1)$.

The low-order app-library interaction information distilled from all the libraries in $N(A_i)$ with Eq. (2) is combined to calculate the overall information for A_i, denoted as $S_{(1)}(A_i)$:

$$S_{(1)}(A_i) = \sum_{L_j \in N(A_i)} \frac{1}{\sqrt{|N(A_i)||N(L_j)|}} S_{(1)}(L_j, A_i) \tag{3}$$

where $\frac{1}{\sqrt{|N(A_i)||N(L_j)|}}$ is the graph Laplacian norm [16, 27] that automatically adjusts the weight of each individual app-library interaction. For example, if mobile app A_i uses a large number of TPLs, it is connected to many library nodes in G and $|N(A_i)|$ will be large. Its corresponding graph Laplacian norm will be low and each TPL in $N(A_i)$ will make a relatively low contribution to $S_{(1)}(A_i)$. The same applies to each TPL in $N(A_i)$. If library L_j is used by many apps, then $|N(L_j)|$ will be large and the corresponding graph Laplacian norm will be low, thus its weight in Eq.(3) will decrease.

Next, vector $\overrightarrow{A_i}_{(1)}$ is obtained which contains the low-order app-library interaction information distilled by the first layer of GNN. It is defined as follows:

$$\overrightarrow{A_i}_{(1)} = LeakyReLU(W_1^{(1)}\overrightarrow{A_i}_{(0)} + S_{(1)}(A_i)) \tag{4}$$

where function LeakyReLU [35] is an activation function widely used in neural networks to control the amount of information transmitted between different layers of a neural network.

Example. *Fig. 5 provides an example process that distills low-order app-library interaction information from TPLs L_1, L_2, and L_3 for mobile app A_1 in Fig. 2. First, GRec employs Eq. (2) to distill individual app-library interaction information from the three corresponding library nodes, i.e., $S_{(1)}(L_1, A_1)$, $S_{(1)}(L_2, A_1)$, and $S_{(1)}(L_3, A_1)$, respectively. Then, it combines the three pieces of information to produce $S_{(1)}(A_1)$ with Eq. (3). Finally, it outputs the latent factor vector $\overrightarrow{A_1}_{(1)}$ with Eq. (4) as the input to Step 2 in Phase 2 and Phase 3.*

Meanwhile, GRec distills low-order app-library interaction information for each library node in G in a similar manner.

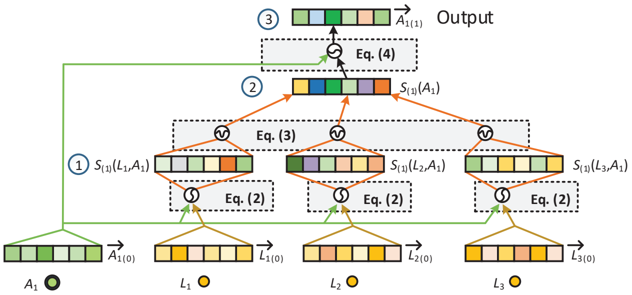
3
$\overrightarrow{A}_{1(1)}$
Output
Eq. (4)
2
$S_{(1)}(A_1)$
Eq. (3)
1
$S_{(1)}(L_1,A_1)$
$S_{(1)}(L_2,A_1)$
$S_{(1)}(L_3,A_1)$
Eq. (2)
Eq. (2)
Eq. (2)
$\overrightarrow{A}_{1(0)}$
$\overrightarrow{L}_{1(0)}$
$\overrightarrow{L}_{2(0)}$
$\overrightarrow{L}_{3(0)}$
A_1
L_1
L_2
L_3

Figure 5: Low-order app-library interaction information distillation for A_1

3.3.2 *Step 2: High-order app-library interaction information distillation.*

In this step, GRec distills high-order app-library interaction information for each node in G. It iterates the process introduced above in Step 1 using the output latent factor vectors produced by the last iteration as input to the next iteration. In this way, information can be distilled from nodes within multiple hops over G for each node in G.

For example, the output vectors of Step 1, i.e., the output vectors of the first layer of the GNN, possess the low-order app-library interaction information distilled from 1-hop neighbor nodes over G. Taking those updated vectors as input, the same process in Step 1 is performed on the second layer of the GNN. Then, the output vectors possess their individual high-order app-library interaction information obtained from their 2-hop neighbor nodes over G. To generalize, the l-th layer of the GNN can distill the high-order app-library interaction information for a node from its l-hop neighbor nodes. Assuming that the GNN has a total of n layers, GRec can distill high-order app-library interaction information for each node from its neighbors within n hops.

In the l-th layer of the GNN, high-order app-library interaction information distilled for app node A_i, denoted as $S_{(l)}(A_i)$, can be recursively defined as:

$$S_{(l)}(A_i) = \sum_{L_j \in N(A_i)} \frac{1}{\sqrt{|N(A_i)||N(L_j)|}} S_{(l)}(L_j, A_i) \tag{5}$$

where $S_{(l)}(L_j, A_i)$ is defined as follows:

$$S_{(l)}(L_j, A_i) = W_1^{(l)}\overrightarrow{L_j}_{(l-1)} + W_2^{(l)}(\overrightarrow{A_i}_{(l-1)} \odot \overrightarrow{L_j}_{(l-1)}) \tag{6}$$

where $W_1^{(l)}, W_2^{(l)} \in \mathbb{R}^{d*d}$ are weight matrices. $\overrightarrow{A_i}_{(l-1)}$ and $\overrightarrow{L_j}_{(l-1)}$ are vectors output by the $(l-1)$-th layer of the GNN.

Similar to Step 1, vector $\overrightarrow{A_i}_{(l)}$ can be obtained based on $S_{(l)}(A_i)$:

$$\overrightarrow{A_i}_{(l)} = LeakyReLU(W_1^{(l)}\overrightarrow{A_i}_{(l-1)} + S_{(l)}(A_i)) \tag{7}$$

Example. *Fig. 6 demonstrates the way GRec distills high-order app-library interaction information from TPL L_5, i.e., node L_5 in Fig. 2, for mobile app A_1, i.e., node A_1 in Fig. 2 along the path L_5-A_3-L_2-A_1 over the app-library graph G when hop = 3. The latent factor vector for node L_5 initialized in Phase 1 is denoted as $\overrightarrow{L_5}_{(0)}$. GRec distills the app-library interaction information from $\overrightarrow{L_5}_{(0)}$ and merges it into vector $\overrightarrow{A_3}_{(1)}$, i.e., the vector for node A_3 updated by the first layer of the GNN. Next, GRec merges the information distilled from*

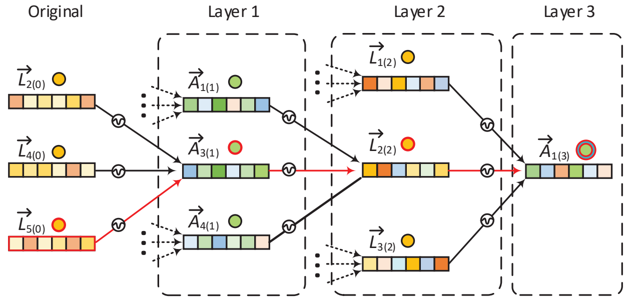
Original
Layer 1
Layer 2
Layer 3
$\overrightarrow{L}_{2(0)}$
$\overrightarrow{L}_{4(0)}$
$\overrightarrow{L}_{5(0)}$
$\overrightarrow{A}_{1(1)}$
$\overrightarrow{A}_{3(1)}$
$\overrightarrow{A}_{4(1)}$
$\overrightarrow{L}_{1(2)}$
$\overrightarrow{L}_{2(2)}$
$\overrightarrow{L}_{3(2)}$
$\overrightarrow{A}_{1(3)}$

Figure 6: High-order app-Library interaction information distillation for A_1

vector $\overrightarrow{A_3}_{(1)}$ into vector $\overrightarrow{L_2}_{(2)}$, i.e., the vector for node L_2 updated by the second layer of the GNN. Finally, GRec merges the information distilled from vector $\overrightarrow{L_2}_{(2)}$ into vector $\overrightarrow{A_1}_{(3)}$. This vector is updated by the third layer of the GNN. In this way, the high-order app-library interaction information possessed by node L_5 is transmitted to A_1.

GRec distills high-order app-library interaction information for each library node L_j via a similar process. We denote the high-order app-library interaction information distilled from l-hop neighbor nodes as $S_{(l)}(L_j)$, and denote the corresponding vector of L_j as $\overrightarrow{L_j}_{(l)}$. Then, there is:

$$S_{(l)}(L_j) = \sum_{A_i \in N(L_j)} \frac{1}{\sqrt{|N(A_i)||N(L_j)|}} S_{(l)}(A_i, L_j) \tag{8}$$

where $S_{(l)}(A_i, L_j)$ is the high-order app-library interaction information produced from $N(L_j)$ for L_j, defined as follows:

$$S_{(l)}(A_i, L_j) = W_1^{(l)} \overrightarrow{A_i}_{(l-1)} + W_2^{(l)} (\overrightarrow{L_j}_{(l-1)} \odot \overrightarrow{A_i}_{(l-1)}) \tag{9}$$

Then, we obtain vector $\overrightarrow{L_j}_{(l)}$ as follows:

$$\overrightarrow{L_j}_{(l)} = LeakyReLU\left(W_1^{(l)} \overrightarrow{L_j}_{(l-1)} + S_{(l)}(L_j)\right) \tag{10}$$

3.4 Phase 3: Aggregation

In Phase 2 (Information Distillation), each layer of the GNN outputs an individual vector for each node in G. Those vectors possess low-order and high-order app-library interaction information distilled from neighbor nodes within different number of hops. In Phase 3 (Aggregation), GRec concatenates all the vectors that belong to the same node to constitute a final vector for each mobile app and TPL. Assuming a total of n layers in the GNN, the constituted latent factor vector for node A_i is formulated as follows:

$$\overrightarrow{A_i^*} = \overrightarrow{A_i}_{(0)} \| \overrightarrow{A_i}_{(1)} \| \overrightarrow{A_i}_{(2)} \| \cdots \| \overrightarrow{A_i}_{(n)} \tag{11}$$

where $\|$ is the concatenation operation.

Similarly, the aggregated latent factor vector for node L_j is defined as follows:

$$\overrightarrow{L_j^*} = \overrightarrow{L_j}_{(0)} \| \overrightarrow{L_j}_{(1)} \| \overrightarrow{L_j}_{(2)} \| \cdots \| \overrightarrow{L_j}_{(n)} \tag{12}$$

By adjusting parameter n, GRec can control the scope of distillation for making recommendations. For example, $n = 1$ means only the low-order app-library interaction information will be distilled, and $n = 5$ means both the low-order and high-order app-library interaction information within 5 hops over G will be distilled. The optimal n can be experimentally obtained.

3.5 Phase 4: Prediction

As introduced in Section 3.2, vector $\overrightarrow{L_j^*}$ models the degree of each latent feature possessed by TPL L_j. Vector $\overrightarrow{A_i^*}$ models the degrees of mobile app A_i's preferences for those latent features. Thus, the potential usefulness of L_j for A_i, denoted by $\hat{R}_(i, j)$, can be approximated as follows:

$$\hat{R}_{i,j} = \overrightarrow{A_i^*} \cdot \overrightarrow{L_j^*} \tag{13}$$

For example, the inner product of $\overrightarrow{A_1^*}$ and $\overrightarrow{L_4^*}$ is the potential usefulness of L_4 for app A_1. When recommending potentially useful TPLs for a mobile app, say A_1, GRec performs the element-wise product on $\overrightarrow{A_1^*}$ with each library's latent factor. Then it obtains the potential usefulness of each library for A_1. Next, it recommends the top nr TPLs with the largest potential usefulness that are not used by A_1 for A_1. The recommended TPLs can be prioritized in developers' search for and evaluation of useful TPLs for A_1.

4 EXPERIMENTAL EVALUATION

We first introduce our experiment settings, then evaluate GRec's performance motivated by four research questions. Finally, we discuss the threats to the validity of the evaluation.

4.1 Experimental Setup

Our experiments are motivated by the following research questions:

- ***RQ1*** *How does GRec perform compared with existing TPL recommendation approaches?*
- ***RQ2*** *Is high-order app-library interaction information useful for improving GRec's performance?*
- ***RQ3*** *Does the dimensionality of the latent space (d) have any impact on GRec's performance?*
- ***RQ4*** *Are GRec's TPL recommendations considered useful by real-world practitioners?*

We implemented GRec using NGCF [36], the state-of-the-art GNN-based recommendation framework. The computer used in the experiments is equipped with Intel i5-7400T CPU, 16 GB RAM, and an NVIDIA Tesla P100 12GB GPU accelerator, running Windows 10 x64 Enterprise, PyTorch 1.3.1, NumPy 1.18.1, SciPy 1.3.2, and Scikit-learn 0.21.3. Our experiments are conducted on the MALib dataset [9], a public real-world dataset that contains 61,722 Android apps, 827 distinct TPLs, and 725,502 app-library usage records. The mobile apps in this dataset were collected from Google Play and the TPL usage records were manually validated. On average, each of the mobile apps in the dataset uses 11.81 TPLs. GRec is designed to recommend potentially useful TPLs for developers who would like to leverage TPLs to improve their mobile apps. Following the same evaluation methodology of [9] and [33], we select mobile apps that use 10 or more TPLs as testing apps for the experiments. The rationale behind this is the same as [9] and [33] - the developers of these mobile apps tend to use TPLs in their mobile apps. In total, 31,432 mobile apps are included in the experiments, using a total of 752 TPLs extracted from 537,011 app-library usage records.

Similar to [9, 23, 33], we mimic real-world scenarios that developers have used a number of TPLs in their mobile apps and are seeking new TPLs to improve their mobile apps further. In the experiments, we employ the cross-validation technique to evaluate GRec and use parameters *rm* and *nr* to simulate various TPL recommendation scenarios. Parameter *rm* determines how many TPLs are removed from each testing app while *nr* determines how many TPLs are recommended for each testing app, i.e, the length of each recommendation list (referred to as *list* hereafter). We set $rm \in \{1, 3, 5\}$ and $nr \in \{5, 10\}$ in different scenarios. In each experiment run, we randomly remove *rm* TPLs from each of the 31,432 testing app. Next, we run GRec to recommend a list with *nr* TPLs for each testing app. Then, we evaluate GRec's performance by inspecting whether those removed TPLs are recommended for corresponding apps. In each experiment run, 31,432 recommendation lists are generated, one for each of the testing apps. Every time a setting parameter varies, we perform 50 experiment runs and report the average results.

We employ the following five metrics for evaluating GRec's recommendation performance. For all the metrics, a higher value indicates higher performance.

- **Mean Precision (MP)** [23, 24, 26]. Given a list, the precision is the ratio of the correctly recommended TPLs over *nr*, i.e., the total number of TPLs in the list. Then, MP is the average precision across all the lists in an experiment run.
- **Mean Recall (MR)** [23, 25, 26, 28, 33]. Given a list, the recall is the ratio of the correctly recommended TPLs in the list over all the TPLs removed from the corresponding testing app. MR is the average recall across all the lists in an experiment run.
- **Mean F1 Score (MF)** [9, 37]. F1 Score conveys a balance between the precision and recall of one list. MF is the average F1 score across all the lists in one experiment run.
- **Mean Average Precision (MAP)** [2, 9, 21]. Given a list with *nr* TPLs, the average precision (AP) measures GRec's ability to put removed TPLs at high positions in the list. It is calculated as follows:

$$AP = \frac{1}{\sum_{i=1}^{nr} cor(i)} \sum_{i=1}^{nr} \frac{\sum_{j=1}^{i} cor(i)}{i} \times cor(i) \tag{14}$$

  where i increases from 1 to *nr* in steps of 1 and $cor(i)$ indicates whether a library at position i is a removed one. It returns 1 if yes and 0 otherwise. MAP is the mean AP across all the lists in one experiment run.
- **Coverage (COV)** [9, 23]. COV measures the diversity of GRec's recommendation results. It is the ratio of distinct TPLs on all the lists in one experiment run over all the distinct TPLs in the MALib dataset. It is an important performance metric for evaluating recommendation approaches in recent years [8], which allows us to inspect whether GRec sacrifices diversity for accuracy.

4.2 *RQ1*: Performance Comparison

We compare GRec to four other approaches, including one baseline approach and three state-of-the-art TPL recommendation approaches.

- **POP** – this always recommends the most popular TPLs not used by the testing app. It is a common baseline for evaluating recommendation approaches [6, 19, 31].
- **LibRec** [33] – this combines association rule mining and collaborative filtering (CF) to make recommendations for conventional Java projects. LibRec has been widely used as a competing approach in recent studies [23, 26, 28].
- **CrossRec** [23] – this approach was proposed very recently and employs the CF-based technique to recommend TPLs for target open-source projects.
- **LibSeek** [9] – this is the state-of-the-art approach that was specifically designed for recommending TPLs for Android apps. It employs a matrix factorization technique to find potentially useful TPLs for mobile apps.

To conduct a fair comparison, the parameter settings of each competing approach are exactly the same as that in [33], [23], and [9], respectively. In GRec, the number of layers in the GNN is 3, i.e., $n = 3$, and the layer size is 128. The size of each latent factor vector is also set to 128, i.e., $d = 128$.

Table 1 compares the average performance of all the competing approaches under different parameter settings. We can see that **GRec achieves the highest performance under all the parameter settings**, indicated by its highest MP, MR, MF, MAP, and COV values. It outperforms POP, LibRec, CrossRec, and LibSeek by 423.79%, 56.66%, 1989.45%, and 29.65%, respectively, on average across all the cases. When $rm = 1$ and $nr = 5$, GRec outperforms POP, LibRec, Crossrec, and LibSeek by 505.12%, 46.75%, 5194.32%, and 33.19%, respectively. When $rm = 5$ and $nr = 10$, it outperforms POP, LibRec, CrossRec, and LibSeek by 358.70%, 62.70%, 747.29%, and 27.12%, respectively.

Compared with POP, LibRec, CrossRec, and LibSeek, the average improvement of GRec is 81.28%, 37.40%, 1908.70%, and 10.33% in MP; 82.74%, 38.56%, 1920.91%, and 11.21% in MR; 81.71%, 37.75%, 1912.35%, and 10.60% in MF; 55.55%, 22.31%, 3564.09%, and 11.65% in MAP, respectively. This demonstrates GRec's superior performance. Surprisingly, **GRec can highly diversify its recommendation results while achieving a high recommendation accuracy**, indicated by its significant advantage in COV over competing approaches, i.e., 1817.66%, 147.28%, 641.20%, and 104.44% against POP, LibRec, CrossRec, and LibSeek, respectively. We find that the COV of POP is particularly low across all the cases. The reason is that POP always recommends a few of the most popular TPLs that have not been used by the testing app. Thus, the other less popular TPLs are seldom recommended. This is a critical limitation as recommending only popular TPLs is not beneficial to developers [9, 14]. In contrast, GRec diversifies the recommendations by recommending both popular and less popular TPLs, indicated by its highest COV values in all cases.

Unlike LibRec, CrossRec, and LibSeek that use only a small portion of low-order app-library interaction information when making recommendations, GRec makes full use of the low-order app-library interaction information and employs also high-order app-library interaction information distilled from the app-library graph in the recommendations. This boosts GRec's performance, indicated by its highest performance in terms of both recommendation accuracy and recommendation diversity.

Table 1: Performance Comparison

Dataset	Approaches	nr=5					nr=10				
		MP	MR	MF	MAP	COV	MP	MR	MF	MAP	COV
rm=1	POP	0.0753	0.3765	0.1255	0.2840	0.0316	0.0457	0.4565	0.0831	0.2949	0.0465
	LibRec	0.1267	0.6335	0.2112	0.4622	0.2921	0.0668	0.6682	0.1215	0.4669	0.2990
	CrossRec	0.0031	0.0155	0.0052	0.0061	0.0472	0.0069	0.0687	0.0125	0.0130	0.0783
	LibSeek	0.1348	0.6741	0.2247	0.5236	0.3346	0.0755	0.7553	0.1373	0.5346	0.3960
	GRec	**0.1521**	**0.7607**	**0.2536**	**0.6269**	**0.6948**	**0.0828**	**0.8283**	**0.1506**	**0.6360**	**0.7918**
rm=3	POP	0.2147	0.3579	0.2684	0.5931	0.0322	0.1341	0.4468	0.2063	0.5682	0.0455
	LibRec	0.2789	0.4648	0.3486	0.6883	0.2916	0.1542	0.5142	0.2373	0.6864	0.2936
	CrossRec	0.0187	0.0312	0.0234	0.0299	0.0896	0.0220	0.0734	0.0339	0.0439	0.1508
	LibSeek	0.3710	0.6183	0.4637	0.7280	0.3245	0.2158	0.7193	0.3320	0.6971	0.3907
	GRec	**0.4099**	**0.6915**	**0.5142**	**0.7977**	**0.6849**	**0.2337**	**0.7879**	**0.3602**	**0.7605**	**0.7824**
rm=5	POP	0.3383	0.3383	0.3383	0.7413	0.0316	0.2180	0.4360	0.2907	0.6813	0.0449
	LibRec	0.4400	0.4400	0.4400	0.6922	0.2885	0.2434	0.4868	0.3245	0.6890	0.2992
	CrossRec	0.0342	0.0342	0.0342	0.0596	0.1701	0.0371	0.0743	0.0495	0.0780	0.2560
	LibSeek	0.5291	0.5291	0.5291	0.7896	0.3141	0.3293	0.6587	0.4391	0.7396	0.3796
	GRec	**0.5868**	**0.5945**	**0.5902**	**0.8397**	**0.6571**	**0.3613**	**0.7312**	**0.4834**	**0.7856**	**0.7536**

4.3 RQ2: Impact of High-Order App-Library Interaction Information

To investigate the usefulness of the high-order app-library interaction information in the recommendations, we vary the number of the layers in the GNN (n) from 1 to 5 in steps of 1 and measure GRec's corresponding performance. When $n = 1$, GRec employs only low-order app-library interaction information to make recommendations. When $n \geq 2$, it employs both low-order and high-order app-library interaction information to make recommendations. Fig. 7 illustrates the impact of the high-order app-library interaction information, where three TPLs are removed from each testing app, i.e., $rm = 3$, and the number of TPLs in each list is 5 and 10, respectively, i.e., $nr \in \{5, 10\}$.

We find that **when n increases from 1 to 2, GRec's performance significantly increases in all the five metrics**. This observation demonstrates the effectiveness of employing high-order app-library interaction information for making TPL recommendations. For example, when $nr = 5$ and $n = 1$, GRec achieves 0.3940, 0.6646, 0.4942, 0.7692, and 0.6209 in MP, MR, MF, MAP, and COV, respectively. When n increases to 2, it achieves 0.4086, 0.6892, 0.5125, 0.7953, and 0.6615 in MP, MR, MF, MAP, and COV, respectively, i.e., 3.70%, 3.71%, 3.70%, 3.40%, and 6.54% higher than when $n = 1$. When n increases to 3, GRec's performance continues to increase, reaching 0.4095, 0.6908, 0.5137, 0.7970, and 0.6811 in MP, MR, MF, MAP, and COV, respectively. When n **continues to increase from 3, GRec's performance decreases slightly in MP, MR, MF, and MAP**. The reason is that an overly large n will include high-order information distilled from many app and library nodes far away from the testing app in G in the recommendations. These apps and TPLs may not be similar to the target app and its TPLs. The noise generated by these nodes lowers GRec's recommendation accuracy. However, GRec's COV value continues to increase, which indicates the effectiveness of leveraging high-order information to increase the diversity.

4.4 RQ3: Impact of Dimensionality of Latent Space

As introduced in Section 3.2, GRec embeds mobile apps and TPLs as d-dimension latent factors to represent their specific features, such as functionality, performance, and compatibility. To study the impact of different values of d on GRec's performance, we vary d from 32 to 512. Fig. 8 shows the experimental results. When d increases, GRec's performance in all the metrics increases. For example, when $nr = 5$, $rm = 3$, and $d = 32$, GRec achieves 0.3646, 0.6153, 0.4573, 0.7439, and 0.5484 in MP, MR, MF, MAP, and COV, respectively. When d increases to 256, GRec achieves 0.4095, 0.6908, 0.5137, 0.7970, and 0.6811 in MP, MR, MF, MAP, and COV, i.e., 12.33%, 12.28%, 12.32%, 7.15%, and 24.22% higher than when $d = 32$. The reason is that a higher dimensionality of the latent space allows GRec to model more potential latent features that reflect the relationships between apps and TPLs. In general, more latent features allow GRec to model the potential usefulness of each TPL for each mobile app more precisely. Thus, **GRec can recommend TPLs more effectively with a higher d.**

Another interesting observation is that, when d increases from 32 to 64 then to 128, GRec's performance increases rapidly, indicated by the increment in all the metrics. However, when d continues to increases from 128 to 256 and then to 512, GRec's performance increase slows down. In practice, a proper value of d can be identified through experiments.

4.5 RQ4: User Study

Our experiments are conducted on testing mobile apps to simulate real-world mobile apps' need for new TPLs. The TPLs removed from testing apps are assumed to be useful for the corresponding testing apps. This is the common assumption made in almost all the research on recommendations in the field of software engineering as well as many other fields. However, the TPLs removed from the testing apps are not necessarily always the best ones. In fact, it is

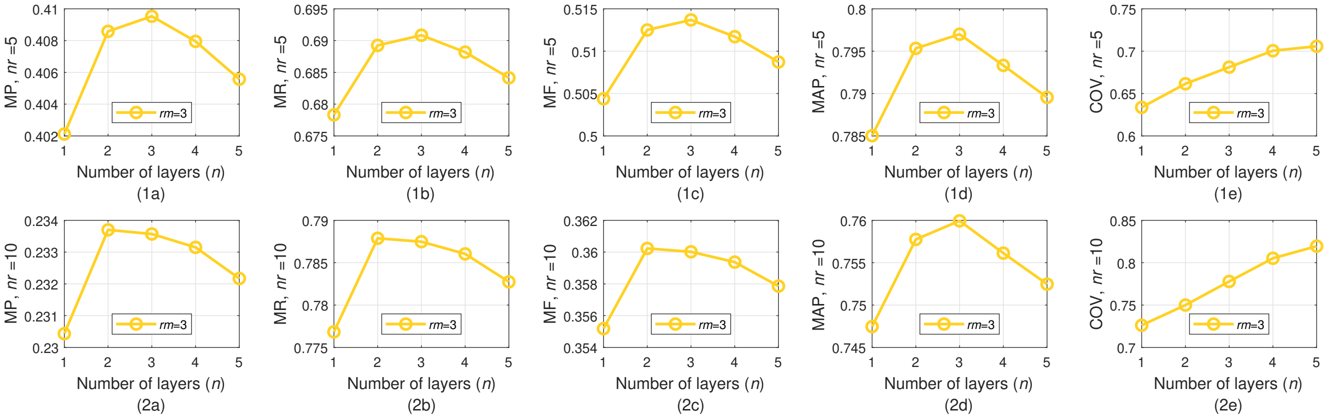

Figure 7: Impact of high-order app-library interaction information

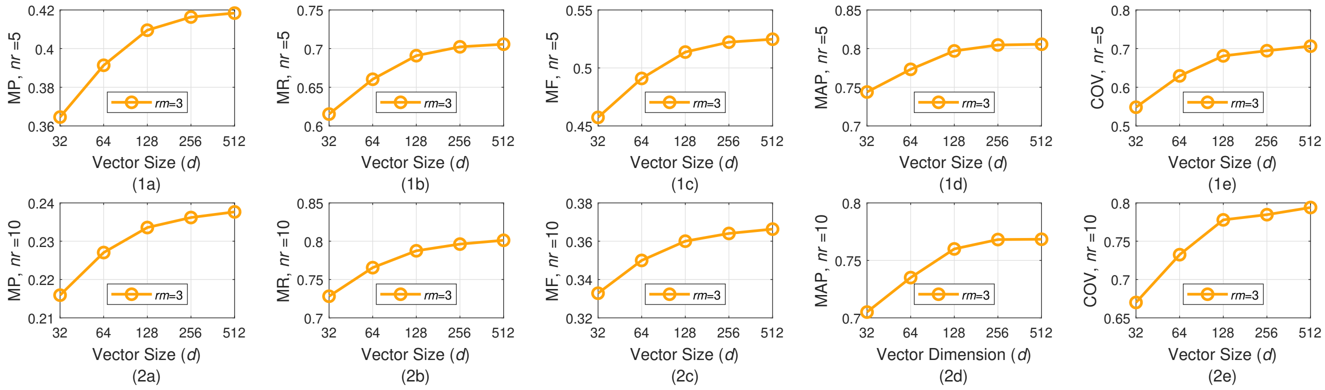

Figure 8: Impact of dimensionality of latent space

impossible to identify the theoretically best TPLs for the testing apps as the ground-truth for the evaluation of GRec. To minimize this threat, we conduct a user study with real-world Android app developers to answer RQ4, i.e., whether GRec's recommendations are indeed useful for real-world mobile apps.

We randomly select 900 Android apps from the F-Droid repository[7], a famous open-source Android app repository. Then, we download the source files of their latest versions as of 22/12/2019, and collect their developer information, i.e., developers' names and emails. Next, we manually inspect the *build.gradle* files of those apps in their source files and obtain all the TPLs used in each mobile app. Then, we run GRec, LibSeek and POP individually based on the MALib dataset to generate three lists of 10 recommended TPLs for each of the 900 apps. Then, we email the lists for each app to the corresponding developers and ask them to rate 1-5 for each TPL in the three lists individually to indicate how much they believe it is useful, e.g., offering useful new features and/or enhancing their apps. Developers rate 1 if they think a TPL is not useful at all and 5 if a TPL is highly useful. To help investigate GRec's impact on different developers, we also inquire about their work experiences. We state that their personal information will be protected.

We sent out 900 emails in total and 74 developers responded. However, 6 of them failed to rate all the recommended TPLs, and thus were excluded from our analysis. Finally, we received 2,040 ratings in total made by 68 developers, i.e., 680 ratings for each of the three approaches. Fig. 9 illustrates the distribution of those ratings. Overall, **GRec receives the highest ratings**, indicated by 263 5s, 329 4s, 72 3s, 13 2s, and 3 1s. 87.06% of the TPLs recommended by GRec are rated 4 or higher. **This demonstrates that most developers highly acknowledge the usefulness of GRec's recommendations**. In contrast, LibSeek received 207 5s, 283 4s, 151 3s, 32 2s, and 7 1s. 72.06% of the TPLs recommended by LibSeek are rated 4 or higher. POP received the lowest ratings overall, i.e., 51 5s, 154 4s, 233 3s, 197 2s, and 45 1s. Almost 70.00% of the TPLs recommended by POP are rated 3 or lower. This observation confirms that recommending only the most popular TPLs is not useful for most developers. The high ratings received by GRec indicate that GRec's recommendations are indeed useful for real-world mobile apps.

Table 2 summarizes developers' ratings for GRec's recommendations according to their work experience. A higher rating indicates higher satisfaction with a recommended TPL. There are in total 11, 28, 17, and 12 developers in the four groups. Through Table 2, we can find that GRec receives higher ratings from developers with less development experience in general. For example, 95.46% of the developers with 2-years work experience or less mark the recommended TPLs 4 or 5, while only 77.50% of the developers with more

[7] https://f-droid.org/en/

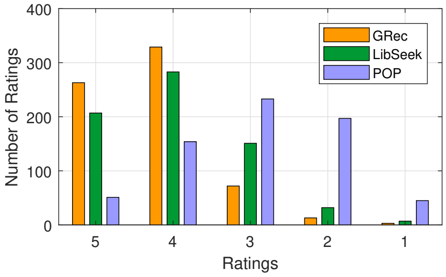

Figure 9: Distribution of developers' feedback

Table 2: Impact of Development Experience on Ratings

Experience	≤2 years	≤4 years	≤6 years	>6 years
# of Developers	11	28	17	12
Rating 5	91	112	33	27
Rating 4	14	147	102	66
Rating 3	5	18	28	21
Rating 2	0	2	7	4
Rating 1	0	1	0	2
Average	4.78	4.32	3.95	3.91

than 6 years work experiences give the same ratings. We can find that experienced developers tend to provide lower ratings overall for all the competing approaches. This observation demonstrates that TPL recommendation is more useful for novice developers.

4.6 Threats to Validity

Internal validity - The main threat is our conclusion on the relationship between GRec's high performance and its utilization of both low-order and high-order app-library interaction information distilled from the app-library graph. To minimize this threat, we varied the number of layers (n) in the experiments. It allows us to observe GRec's performance when it makes recommendations with and without high-order app-library interaction information. The optimal n can be experimentally obtained for GRec to achieve the best performance.

External validity - The main threat comes from the scale of the user study. Only 74 mobile app developers participated in our user study. However, the results of the user study are consistent with our experimental results. In the future, to mitigate the threat, we plan to perform a large-scale user study. The second main threat comes from the reuse of the MALib dataset. Although it has been carefully inspected [9], it may still contain errors. In the future, we plan to perform experiments on a larger dataset to further evaluate GRec's performance.

Construct validity - The main threat comes from the four competing approaches used in our experiments. These approaches leverage only limited low-order app-library interaction information mined from the MALib dataset to make TPL recommendations. Thus, their performance tend to be lower than GRec. To minimize this threat, we varied rm, nr, hop, and d to evaluate GRec's performance comprehensively. Thus, this threat is valid but not significant.

5 RELATED WORK

Recommendation techniques have been widely used to facilitate software development [3, 10, 40, 42, 46]. A number of approaches has been proposed for recommending program clips or APIs of particular TPLs to improve development efficiency. To name a few, Zheng et al. [47] propose an approach that recommends new APIs for API replacement during software development. Thung et al. take the textual description of a feature request as input, and recommend potentially useful methods (APIs) to help developers implement the feature [34]. Besides, they consider also the similarity between two features based on the number of similar API methods used for implementing the two features. Huang et al. employ word embedding technique to bridge the gap between demand descriptions and structured API descriptions when recommending APIs for software development [13]. Liu et al. propose RecRank to improve the top-1 API recommendation accuracy based on the API usage paths in the corresponding call graph [20]. Xie et al. distill hierarchical context information from project-specific code by analyzing its call graph, then recommend new APIs for development [38]. Nguyen et al. employ app-library interaction filtering (CF) to recommend APIs for open-source projects [24]. A major difference between GRec and the above approaches is that GRec recommends as a whole library rather than specific program clips or APIs. Besides, GRec requires app-library usage records to make the recommendations without the need for extra contextual information.

In recent years, several approaches are proposed to help developers find potential useful TPLs through mining app-library usage patterns [5, 26, 28, 29, 32]. Saied et al. propose COUPminer that employs both client-based mining and library-based usage mining to cooperatively mine app-library usage patterns [29]. Ouni et al. propose a search-based approach, namely LibFinder, to detect relevant TPLs for software maintenance and evolution [26]. Specifically, they employ the semantic similarity between source codes and TPL co-usage relationship to mine app-library usage patterns. Recently, another tool named LibCUP is proposed by Saied et al. for mining app-library usage patterns [28]. LibCUP computes the similarity between different TPLs based on their usage history. Then, it applies a multi-layer clustering approach to categorize different TPLs. Chouchen et al. employed non-dominated sorting genetic algorithm to recommend TPLs by considering TPL co-usage information, TPL functional diversity, and app ratings [5].

Inspired by the great success of CF in a variety of recommendation domains, researchers have started to employ CF to recommend TPLs for software development in recent years. The advantage of CF-based approaches is that they can model app-library usage patterns in a latent way. They make recommendations based on only similar app-library usage in software projects without having to explicitly model TPLs' functionality, reliability, compatibility, and dependency. LibRec is the first approach that employs CF to recommend TPLs for Java projects [33]. It combines association rule mining and collaborative filtering to mine app-library usage patterns and then recommends potentially useful TPLs for target projects. Similarly, CrossRec recommends TPLs for open-source software projects based on collaborative filtering [23]. LibSeek is the first tool specifically designed for recommending TPLs for Android app development [9]. It employs the matrix factorization technique to find potentially useful TPLs and adaptively adjusts the weights of different TPLs to diversify recommendation results. However, these existing tools exploit only limited low-order information extracted from app-library usage records, which prevents them from providing highly accurate recommendations. In this paper, we propose GRec, which takes a giant step to advance TPL recommendation for mobile apps significantly and innovatively with a graph neural network (GNN). GRec models app-library interactions with an app-library graph, then distills both low-order and high-order app-library interaction information with the GNN for making TPL recommendations. Its performance is compared against a baseline popularity-based recommendation approach and three state-of-the-art approaches including LibRec, CrossRec, and LibSeek, via experiments and a user study. As described in Section 4.2, GRec significantly outperforms all the competiting approaches in terms of both accuracy and diversity.

6 CONCLUSION AND FUTURE WORK

We have proposed GRec, a novel Graph Neural Network (GNN)-based approach for recommending potentially useful third-party libraries (TPLs) for mobile app development. GRec can help relieve developers’ burden in searching for and evaluating useful TPLs for improving their mobile apps. Unlike existing tools that use only limited low-order app-library interaction information, GRec models the relationships between mobile apps and TPLs into an app-library graph. Then, it distills both low-order and high-order app-library interaction information with a GNN to make TPL recommendations. The experimental results on 31,432 Android apps and the user study demonstrate the high performance of GRec.

In future, we will study how to make recommendations for specific versions of TPLs. We also plan to perform experiments on a larger dataset, and conduct a user study with more participants.

ACKNOWLEDGEMENT

This work is partly funded by Australian Research Council Discovery Projects DP180100212, DP200102491, and DP200100020. John Grundy is funded by ARC Laureate Fellowship FL190100035. Li Li is funded by ARC Discovery Early Career Researcher Award DE200100016. Qiang He is the corresponding author of this paper.

REFERENCES

[1] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 356–367. https://doi.org/10.1145/2976749.2978333

[2] Lingfeng Bao, Xin Xia, David Lo, and Gail C Murphy. 2019. A large scale study of long-time contributor prediction for GitHub projects. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2918536

[3] Eiji Adachi Barbosa and Alessandro Garcia. 2017. Global-aware recommendations for repairing violations in exception handling. *IEEE Transactions on Software Engineering* 44, 9 (2017), 855–873. https://doi.org/10.1145/3180155.3182539

[4] R. Bell, Y. Koren, and C. Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42 (08 2009), 30–37. https://doi.org/10.1109/MC.2009.263

[5] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. 2020. AndroLib: Third-party software library recommendation for Android applications. In *International Conference on Software and Software Reuse*. Springer, 208–225. https://doi.org/10.1007/978-3-030-64694-3_13

[6] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. 2007. Google news personalization: Scalable online collaborative filtering. In *16th International Conference on World Wide Web*. ACM, 271–280. https://doi.org/10.1145/1242572.1242610

[7] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on android. In *2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2187–2200. https://doi.org/10.1145/3133956.3134059

[8] Mouzhi Ge, Carla Delgado-Battenfeld, and Dietmar Jannach. 2010. Beyond accuracy: Evaluating recommender systems by coverage and serendipity. In *4th ACM Conference on Recommender Systems*. ACM, 257–260. https://doi.org/10.1145/1864708.1864761

[9] Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, and Yun Yang. 2020. Diversified third-party library prediction for mobile app development. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.2982154

[10] Qiang He, Rui Zhou, Xuyun Zhang, Yanchun Wang, Dayong Ye, Feifei Chen, John C. Grundy, and Yun Yang. 2017. Keyword search for building service-based systems. *IEEE Transactions on Software Engineering* 43, 7 (2017), 658–674. https://doi.org/10.1109/TSE.2016.2624293

[11] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*. 173–182. https://doi.org/10.1145/3038912.3052569

[12] Thomas Hofmann. 2004. Latent Semantic Models for Collaborative Filtering. *ACM Transactions on Information Systems* 22, 1 (Jan. 2004), 89–115. https://doi.org/10.1145/963770.963774

[13] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 293–304. https://doi.org/10.1145/3238147.3238191

[14] Marius Kaminskas and Derek Bridge. 2017. Diversity, serendipity, novelty, and coverage: A survey and empirical analysis of beyond-accuracy objectives in recommender systems. *ACM Transactions on Interactive Intelligent Systems* 7, 1 (2017), 2. https://doi.org/10.1145/2926720

[15] Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2019. Mimic: UI compatibility testing system for Android apps. In *41st International Conference on Software Engineering (ICSE)*. IEEE, 246–256. https://doi.org/10.1109/ICSE.2019.00040

[16] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[17] Maxime Lamothe and Weiyi Shang. 2020. When APIs are intentionally bypassed: An exploratory study of API workarounds. In *42nd International Conference on Software Engineering (ICSE)*, Vol. 2020.

[18] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. 2017. LibD: Scalable and precise third-party library detection in Android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 335–346. https://doi.org/10.1109/ICSE.2017.38

[19] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing* 1 (2003), 76–80. https://doi.org/10.1109/MIC.2003.1167344

[20] Xiaoyu Liu, LiGuo Huang, and Vincent Ng. 2018. Effective API recommendation without historical software repositories. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 282–292. https://doi.org/10.1145/3238147.3238216

[21] Zhongxin Liu, Xin Xia, David Lo, and John Grundy. 2019. Automatic, highly accurate app permission recommendation. *Automated Software Engineering* 26, 2 (2019), 241–274. https://doi.org/10.1007/s10515-019-00254-6

[22] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and accurate detection of third-party libraries in Android apps. In *38th International Conference on Software Engineering Companion (ICSE)*. ACM, 653–656. https://doi.org/10.1145/2889160.2889178

[23] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. CrossRec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software* 161 (2020), 110460. https://doi.org/10.1016/j.jss.2019.110460

[24] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. Focus: A recommender system for mining API function calls and usage patterns. In *41st International Conference on Software Engineering (ICSE)*. IEEE, 1050–1060. https://doi.org/10.1109/ICSE.2019.00109

[25] Chao Ni, Xin Xia, David Lo, Xiang Chen, and Qing Gu. 2020. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3001739

[26] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M German, and Katsuro Inoue. 2017. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology* 83 (2017), 55–75. https://doi.org/10.1016/j.infsof.2016.11.007

[27] Jiezhong Qiu, Jian Tang, Hao Ma, Yuxiao Dong, Kuansan Wang, and Jie Tang. 2018. Deepinf: Social influence prediction with deep learning. In *24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2110–2119. https://doi.org/10.1145/3219819.3220077

[28] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. 2018. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software* 145 (2018), 164–179. https://doi.org/10.1016/j.jss.2018.08.032

[29] Mohamed Aymen Saied and Houari Sahraoui. 2016. A cooperative approach for combining client-based and library-based API usage pattern mining. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10. https://doi.org/10.1109/ICPC.2016.7503717

[30] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Filomena Ferrucci. 2020. Third-party libraries in mobile apps. *Empirical Software Engineering* 25, 3 (2020), 2341–2377. https://doi.org/10.1007/s10664-019-09754-1

[31] Yue Shi, Martha Larson, and Alan Hanjalic. 2014. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *Comput. Surveys* 47, 1 (May 2014), 3:1–3:45. https://doi.org/10.1145/2556270

[32] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2013. Automatic discovery of function mappings between similar libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 192–201. https://doi.org/10.1109/WCRE.2013.6671294

[33] F. Thung, D. Lo, and J. Lawall. 2013. Automated library recommendation. In *20th Working Conference on Reverse Engineering (WCRE)*. 182–191. https://doi.org/10.1109/WCRE.2013.6671293

[34] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. 2013. Automatic recommendation of API methods from feature requests. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 290–300. https://doi.org/10.1109/ASE.2013.6693088

[35] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. 2019. KGAT: Knowledge graph attention network for recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 950–958. https://doi.org/10.1145/3292500.3330989

[36] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural graph collaborative filtering. In *Proceedings of the 42nd Lnternational ACM SIGIR Conference on Research and Development in Information Retrieval*. 165–174. https://doi.org/10.1145/3331184.3331267

[37] Yanchun Wang, Qiang He, Dayong Ye, and Yun Yang. 2018. Formulating criticality-based cost-effective fault tolerance strategies for multi-tenant service-based systems. *IEEE Transactions on Software Engineering* 44, 3 (2018), 291–307. https://doi.org/10.1109/TSE.2017.2681667

[38] Rensong Xie, Xianglong Kong, Lulu Wang, Ying Zhou, and Bixin Li. 2019. HiRec: API Recommendation using Hierarchical Context. In *30th International Symposium on Software Reliability Engineering*. IEEE, 369–379. https://doi.org/10.1109/ISSRE.2019.00044

[39] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. 2020. Why reinventing the wheels? An empirical study on library reuse and re-implementation. *Empirical Software Engineering* 25, 1 (2020), 755–789. https://doi.org/10.1007/s10664-019-09771-0

[40] Congying Xu, Bosen Min, Xiaobing Sun, Jiajun Hu, Bin Li, and Yucong Duan. 2019. MULAPI: A tool for API method and usage location recommendation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*. IEEE, 119–122. https://doi.org/10.1109/ICSE-Companion.2019.00053

[41] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*.

[42] Luting Ye, Hailong Sun, Xu Wang, and Jiaruijue Wang. 2018. Personalized teammate recommendation for crowdsourced software developers. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 808–813. https://doi.org/10.1145/3238147.3240472

[43] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2020. Automated third-party library detection for Android applications: Are we there yet?. In *The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*.

[44] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *Comput. Surveys* 52, 1 (2019), 1–38. https://doi.org/10.1145/3285029

[45] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 141–152. https://doi.org/10.1109/SANER.2018.8330204

[46] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. ReCDroid: Automatically reproducing android application crashes from bug reports. In *41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139. https://doi.org/10.1109/ICSE.2019.00030

[47] Wujie Zheng, Qirun Zhang, and Michael Lyu. 2011. Cross-library API recommendation using web search engines. In *19th ACM SIGSOFT Symposium and 13th European Conference on Foundations of Software Engineering*. ACM, 480–483. https://doi.org/10.1145/2025113.2025197