Neural Library Recommendation by Embedding Project-Library Knowledge Graph

Bo Li, Haowei Quan, Jiawei Wang, Pei Liu, Haipeng Cai, Yun Yang, and Li Li

Abstract—The prosperity of software applications brings fierce market competition to developers. Employing third-party libraries (TPLs) to add new features to projects under development and to reduce the time to market has become a popular way in the community. However, given the tremendous TPLs ready for use, it is challenging for developers to effectively and efficiently identify the most suitable TPLs. To tackle this obstacle, we propose an innovative approach named PyRec to recommend potentially useful TPLs to developers for their projects. Taking Python project development as a use case, PyRec embeds Python projects, TPLs, contextual information, and relations between those entities into a knowledge graph. Then, it employs a graph neural network to capture useful information from the graph to make TPL recommendations. Different from existing approaches, PyRec can make full use of not only project-library interaction information but also contextual information to make more accurate TPL recommendations. Comprehensive evaluations are conducted based on 12,421 Python projects involving 963 TPLs, 9,675 extra entities, 121,474 library usage records, and 73,277 contextual records. Compared with five representative approaches, PyRec improves the recommendation performance by 35.81% to 198.87% on average across all cases.

Index Terms—third-party library, recommendation, knowledge graph, graph neural network, Python

1 Introduction

Recent years have witnessed the astonishing growth of software applications, especially open-source Python applications. As reported by IEEE Spectrum [1], Python has become the most popular language since 2021. Many popular applications like Google search engine, YouTube, and Instagram are built in Python [2]. One reason that fuels Python's popularity could be the large number of third-party libraries (TPLs) readily to be used by the community [3]. For example, more than 390,000 Python TPLs with over 3 million versions are available in May 2023 in the Python Package Index (PyPI) repository [4].

Compared with programming from scratch, TPLs offer tailor-made APIs with the same functionalities [5] but less bugs/deficiencies [6]. Therefore, seeking TPLs with desired functionalities and integrating them in projects under development is much more effective [7]–[10]. Indeed, it has become a common practice for developers to regularly use TPLs to accelerate their development process and/or deliver new features [11].

Unfortunately, given the huge number of TPLs available for use, it is challenging for developers to seek the most suitable TPLs for their projects [12], [13]. First, manually inspecting the functionalities, interfaces, performance, etc., of tremendous TPLs is very time-consuming [14]. It is even more sophisticated nowadays as TPLs are evolving rapidly [2] and the time-to-market constraint is becoming tighter [15]. Second, TPL usage has specific characteristics [11], [16], e.g., combinations and dependencies. Finding appropriate TPLs fulfilling such characteristics is another time-consuming process [5], [6].

Inspired by the great success of recommender systems in a variety of domains [17], many TPL recommendation approaches have been proposed recently to accelerate the TPL seeking process [6], [15], [18], [19]. Generally, they provide developers with a short list of TPLs for consideration. For example, LibRec [18] and CrossRec [6] are collaborative filtering (CF)-based approaches that find potentially useful TPLs for Java projects. The general idea is to recommend TPLs used by similar projects but not yet by the current project. LibSeek [19] is designed for recommending TPLs for Android mobile apps. It embeds features of mobile apps and TPLs into latent vectors via matrix factorization (MF) to find potentially useful TPLs for a given Android mobile app. GRec [15] is a deep learning (DL)-based approach that recommends TPLs for Android apps. It models mobile apps, TPLs, and their usage relations as a bipartite graph, and then employs the graph neural network (GNN) to distill information from the graph to improve the recommendation performance.

Preliminary user studies have confirmed the usefulness and effectiveness of recommending TPLs for software application development [15], [19]. However, the performance of existing approaches needs to be further improved. Specifically, they treat different projects or TPLs as independent instances and utilize only *project-library interaction information*, i.e., which project has used which TPLs, to make recom-

- *Bo Li is with the College of Arts, Business, Law Education, and Information Technology, Victoria University, Melbourne, AU, 3011.E-mail: li.bo@vu.edu.au*
- *Haowei Quan, Jiawei Wang, and Pei Liu are with the Faculty of Information Technology, Monash University, Melbourne, AU, 3800. E-mail: {haowei.quan, jiawei.wang1, pei.liu}@monash.edu*
- *Haipeng Cai is with the School of Electrical Engineering and Computer Science (EECS) at Washington State University, Pullman, USA, 99163.E-mail: haipeng.cai@wsu.edu*
- *Yun Yang is with the Department of Computing Technologies, Swinburne University of Technology, Melbourne, VIC, Australia, 3122. E-mail: yyang@swin.edu.au*
- *Li Li is with the School of Software, Beihang University, Beijing, China, 100191.E-mail: lilicoding@ieee.org*

Manuscript received xxx xx, 2023; revised xxx xx, 2023.

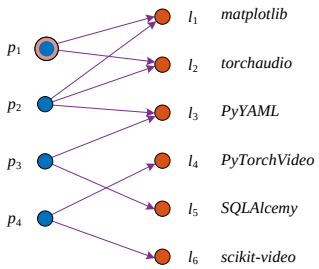

Fig. 1. Exemplar Project-Library Bipartite Graph $\mathcal{BG}$

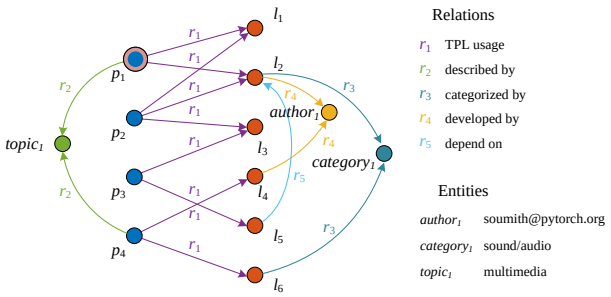

Fig. 2. Exemplar Project-Library Knowledge Graph $\mathcal{KG}$

mendations. This is acceptable when the developers have determined many TPLs to be used in a project. However, when the project is at an early development stage, usually only very limited TPLs have been determined. In this case, there would be less project-library interaction information available for recommendation. As a result, the recommendation performance is much lower.

A potential solution is to utilize contextual information [20] like the inherent relations between different projects and different TPLs, which have been overlooked by existing approaches. For example, projects with the same topic [21] on GitHub may share similar characteristics implemented by the same TPL. TPLs with the same keywords in a category may have the same/similar functionalities and interfaces and are exchangeable to each other [15]. In addition, a TPL usually depends on some other TPLs and adds more features to the dependency libraries. Once a dependency TPL is chosen by a project, the TPLs depending on it may also be of interest to developers. In practice, the contextual information, including the above-mentioned ones, are helpful in finding more suitable TPLs [22]. However, existing approaches have unfortunately ignored such information and thus their performance are constrained.

In this paper, we take TPL recommendation for Python projects as a use case. The reason is that Python has emerged as the most popular programming language in recent years. Both the amount of Python projects and the amount of Python libraries ready for use are much greater than the other programming languages [23]. In addition, Python developers rely on the rich functionalities offered by the huge collection of TPLs for fast prototyping [2], [23], [24]. Therefore recommending suitable TPLs can be beneficial for them. However, the TPL recommendation for Python projects has been neglected by the community. *Please note that the approach proposed in this paper, i.e., PyRec, can also be applied to other programming scenarios, like recommending TPLs for Android mobile app development or conventional Java project development, wherever the required contextual information is available.*

PyRec is an innovative approach that makes full use of both project-library interaction information and contextual information to provide high-accuracy recommendations. Specifically, we map the Python projects, TPLs, and their interactions into a *bipartite graph* (BG) in which Python projects and TPLs are nodes and the project-library interactions are edges connecting each pair of nodes, as exemplified in Fig. 1. Then, we extend BG to a *knowledge graph* (KG) by adding new nodes and edges according to available contextual information, as illustrated in Fig. 2. Next, we employ GNN to distill useful information from the KG to make TPL recommendations. Compared with the state-of-the-art approaches like LibRec [18], CrossRec [6], LibSeek [19], and GRec [15], PyRec can make more accurate recommendations, with an average accuracy improvement of 87.39%, 198.87%, 35.81%, and 48.93%, respectively. The key contributions of this research are concluded as follows.

- We are the first to use both project-library interaction information and contextual information for TPL recommendation.
- With the help of contextual information, we model Python projects, TPLs, and their interactions as a knowledge graph (KG), in which inherent relations between different projects and different TPLs are represented. This allows capturing more information crucial for accurate TPL recommendation.
- We propose an innovative GNN-based DL model to distill useful information from the generated KG for TPL recommendation. In addition, with a dedicated attention mechanism, our model can automatically identify the usefulness of information possessed by different neighbor nodes and relations and thus can distill more useful information and mitigate the negative impact of unuseful information.
- We make the first attempt to recommend potentially useful TPLs for Python projects. We prototype PyRec and conduct extensive experiments on a large-scale dataset including 12,421 Python projects, 963 distinct TPLs, 9,675 extra entities, 121,474 project-library interaction records, and 73,277 pieces of contextual information.

The rest of this paper is organized as follows. Section 2 motivates the research of this paper. Section 3 introduces PyRec in detail. Section 4 evaluates PyRec experimentally. Section 5 reviews related work. Then, Section 6 concludes this paper and points out future work.

2 Motivating Example

Fig. 1 provides an exemplar bipartite graph (denoted as $\mathcal{BG}$) modeling the project-library interactions. Specifically, it has 4 Python project nodes denoted as $p_1, p_2, ..., p_4$, respectively, and 6 TPL nodes denoted as $l_1, l_2, ..., l_6$, respectively. The direct project-library interactions are represented by edges between the corresponding projects and TPLs. For example, the edge between p_1 and l_2 (*torchaudio* [25]) indicates that Python project p_1 uses TPL *torchaudio*. Let us take seeking new TPLs for project p_1 as an example hereafter.

CF-based approaches like CrossRec [6] make TPL recommendations based on the similarities between different projects in terms of TPL usage. For example, project p_1 invokes two TPLs, i.e., l_1 (*matplotlib* [26]) and l_2 (*torchaudio*). In the meantime, project p_2 also invokes these two TPLs. This indicates that p_1 and p_2 have some similarities in terms of TPL usage. In this case, TPLs used in p_2 but not yet in p_1 may be of interest to p_1, such as l_3 (*PyYAML* [27]). Thus, *PyYAML* is recommended to p_1. Apparently, CF-based approaches only utilize part of those direct project-library interactions in $\mathcal{BG}$, i.e., interactions involved in projects similar to p_1.

MF-based approaches like LibSeek [19] embed projects and TPLs into latent vectors to make recommendations. They utilize all direct project-library interactions in $\mathcal{BG}$ to learn those latent vectors. For example, p_1's latent vector is learned based on two interactions, i.e., interaction between p_1 and l_1 (*matplotlib*) and interaction between p_1 and l_2 (*torchaudio*). Similarly, l_1's latent vector is learned based on the interaction between p_1 and l_1, and the interaction between p_2 and l_1.

In recent years, deep learning has been widely adopted to solve a variety of technical problems. Many DL-based recommender systems have been proposed. However, as the recommender systems are usually domain-specific [28], general DL-based systems like ChatGPT[1] could not perform well in the TPL recommendation field. Therefore, specifically designed DL-based recommendation approaches are in urgent need by the software engineering (SE) community. GRec [15] is the first DL-based approach that is capable of exploiting transitive information in $\mathcal{BG}$ to learn the latent vectors. For example, both projects p_1 and p_2 use l_1 (matplotlib) and l_2 (torchaudio), and thus p_2 is similar to p_1. Similarly, both projects p_2 and p_3 use l_3 (*PyYAML*), and thus p_3 is similar to p_2. In this case, although p_3 is not similar to p_1, it may contribute useful information to the learning of p_1's latent vector. This transitive relationship between p_3 and p_1 is called *high-order interaction* [15] reflected by the path p_3-l_3-p_2-$l_2(l_1)$-p_1 in $\mathcal{BG}$. The utilization of transitive information boosts GRec's TPL recommendation performance.

Theoretically, all the above-mentioned approaches treat projects and TPLs as independent instances, i.e., neither any relationships between Python projects nor any relationships between TPLs are considered. Therefore, they make TPL recommendations based solely on the direct/transitive project-library interactions possessed by $\mathcal{BG}$. Unfortunately, the overlook of real-world project relations and TPL relations inevitably undermined their TPL recommendation performance. Indeed, such relations can be identified based on contextual information relevant to Python projects or TPLs. For example, the information on developers, categories, introductions, and keywords can help identify the relationships between two TPLs. As demonstrated by **Fig. 2**, projects p_1 and p_2 are described by the same topic keyword $topic_1$ (*multimedia*) on GitHub, and thus have similar functionalities. Therefore, p_4 may contribute to the learning of p_1's latent vector. However, it is not utilized by existing approaches (e.g., there is no interaction between p_1 and p_4 represented in Fig. 1). This also applies to TPLs l_4 (*PyTorchVideo* [29]) and l_6 (*scikit-sound* [30]).

To model these project relations and TPL relations, we add new entities, e.g., authors and keywords, into $\mathcal{BG}$. Then, we create edges between the original project/TPL nodes and newly added entity nodes. In this case, $\mathcal{BG}$ is converted to KG (denoted as $\mathcal{KG}$) shown in Fig. 2. Different from $\mathcal{BG}$ in which all edges have the same type, $\mathcal{KG}$ has different types of edges. For example, an edge from p_1 to l_1 represents the TPL usage interactions (denoted as r_1), the edge from l_2 to entity $category_1$ (*sound/audio*) indicates that *torchaudio* belongs to category *sound/audio* on PyPI (denoted as r_3), etc. A unique characteristic of TPLs is that they could depend on other TPLs. To model such dependencies, we add new edges for each pair of involved TPLs in $\mathcal{BG}$. For example, l_5 (*SQLAlchemy* [31]) depends on l_2 (*torchaudio*), we have an edge from l_5 to l_3 with the type of r_5.

With $\mathcal{KG}$, we can mine more information beneficial for TPL recommendation for p_1. For example, l_2 (*torchaudio*) and l_4 (*PyTorchVideo*) are developed by the same developer *soumith*. p_1 may also have interests in *PytorchVideo* when it uses *torchaudio*. Then, useful information can be distilled from l_4 for p_1 through the path $PytorchVideo \xrightarrow{developed\ by} soumith \xrightarrow{develop} torchaudio \xrightarrow{TPL\ usage} p_1$. Similarly, as l_6 (*scikit-sound*) and l_2 (torchaudio) belong to the same category *sound/audio* on PyPI, l_6 may also contribute useful information to p_1. This can be captured through the path $scikit-sound \xrightarrow{categorized\ by} sound/audio \xrightarrow{categorize} torchaudio \xrightarrow{TPL\ usage} p_1$. Besides, as l_5 (*SQLAlchemy*) is implemented based on *torchaudio*, it may provide new features for p_1 and thus of interests to p_1. This information can be represented through the path $SQLAlchemy \xrightarrow{depend\ on} torchaudio \xrightarrow{TPL\ usage} p_1$. We can find that extending $\mathcal{GB}$ to $\mathcal{KG}$ with contextual information can help utilize more information that has been overlooked by existing approaches.

However, it is challenging to incorporate contextual information into TPL recommendations. Indeed, there are many significant differences between implementing recommender systems based on $\mathcal{KG}$, e.g., PyRec, and implementing recommender systems based on $\mathcal{BG}$, e.g., GRec. First, there are much more nodes and edges in $\mathcal{KG}$ than that in $\mathcal{BG}$, determined by the total number of entities involved in the contextual information. The $\mathcal{KG}$-based recommendation approach should be able to handle those extra and usually large volumes of nodes and edges. Second, different from $\mathcal{BG}$ which has only project nodes and library nodes, $\mathcal{KG}$ contains many different types of nodes, determined by the types of involved entities in the contextual information. Hence, the $\mathcal{KG}$-based recommendation approach should be capable of handling more node types. Third, all edges in $\mathcal{BG}$ have the same type and thus such edges do not need to be embedded by the DL model. In contrast, edges in $\mathcal{KG}$ have many different types. Therefore, the $\mathcal{KG}$-based recommendation approach must embed those edges by the DL model. Fourth, nodes in different paths in $\mathcal{KG}$ usually do not contribute information evenly. For example, l_4, l_5, and l_6 all connect to l_2. However, l_4 connects to l_2 as they have the same developer $soumith$, l_6 connects to l_2 as they are in the same category $sound/audio$, and l_5 connects to

1. https://openai.com/blog/chatgpt

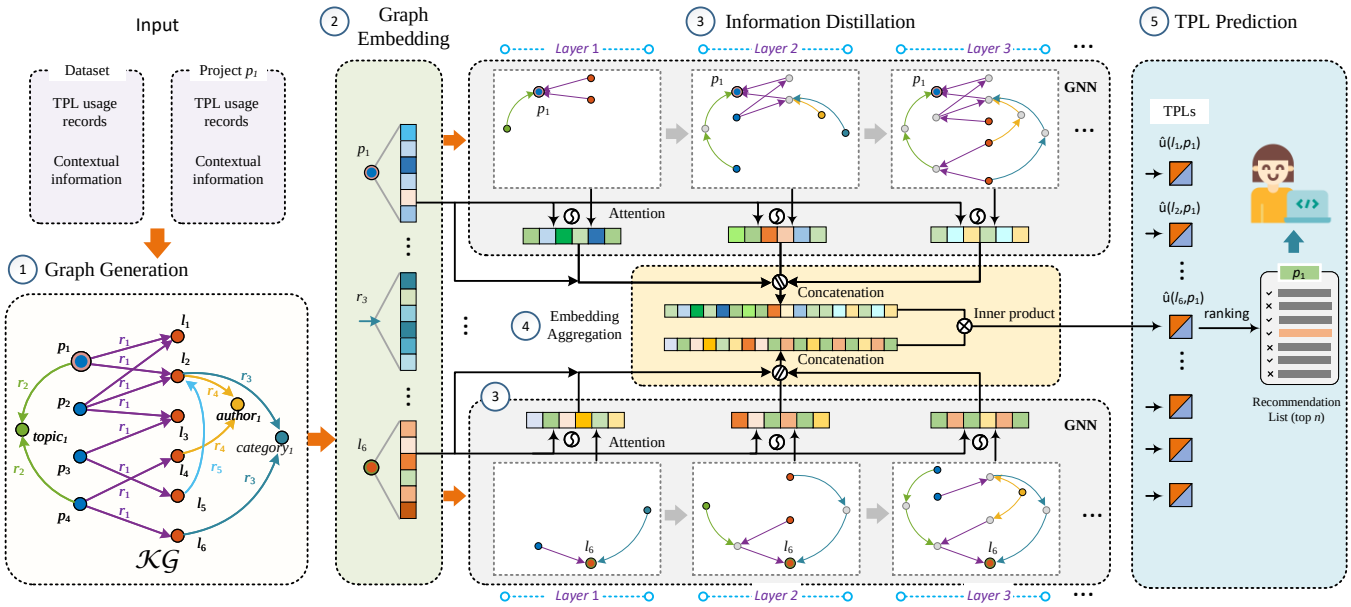

Fig. 3. General process of PyRec

l_2 due to the dependency relationship. As there are many different types of nodes and edges, it is hard to empirically set up their weights, i.e., how much information a node can contribute through a specific path in $\mathcal{KG}$. Therefore, new attention mechanisms are needed to automatically formulate the usefulness of different types of relations. Finally, at the model optimization stage, in addition to optimizing the project-library interactions, we also need to optimize the embeddings of extra nodes and all edges, which is referred to as Graph Embedding Optimization in this paper. Thus, a new model optimization strategy is needed.

To summarise, new approaches that can make precise use of both contextual information and project-library interaction information are needed by the SE community to help developers effectively find useful TPLs.

3 PyRec Approach

3.1 Process Overview

Given a Python project, say p_1 in Fig. 2, PyRec takes three pieces of data as input, including TPL usage records of existing projects, p_1's current TPLs, and contextual information, i.e., project-project and TPL-TPL relationships. It goes through five phases to recommend potentially useful TPLs for p_1, as shown in **Fig. 3**. In **Phase 1 (Graph Generation)**, PyRec builds up $\mathcal{KG}$ based on given TPL usage records and contextual information. Different from GRec [15] that has only project nodes and TPL nodes, PyRec identifies new entity nodes, e.g., $topic_1$, $author_1$, and $category_1$ in Fig. 2, based on contextual information. Then, it creates edges between those entity nodes and existing project/TPL nodes to supplement $\mathcal{KG}$. In **Phase 2 (Graph Embedding)**, PyRec embeds each node and edge in $\mathcal{KG}$ into an individual latent vector. This is different from GRec [15] in which only nodes are embedded. In **Phase 3 (Information Distillation)**, PyRec employs a multi-layer GNN to distill useful information from $\mathcal{KG}$. Specifically, it uses the first GNN layer to distill information from neighbor nodes one hop away, it uses the second layer to distill information from neighbor nodes two hop away, and so on. PyRec implements unique attention mechanisms to help identify more useful information. With a m-layer GNN, PyRec can eventually explore useful information for p_1 from its neighbor nodes within m hops in the $\mathcal{KG}$. In **Phase 4 (Embedding Aggregation)**, for each project/TPL node, its latent vector and information collected by GNN are concatenated into a new vector. Finally, after training in **Phase 5 (TPL Prediction)**, PyRec predicts the usefulness of each TPL to p_1 and recommends top n the most useful TPLs. Different from GRec [15], the training process of PyRec consists of two parts: graph embedding optimization and project-library interaction optimization.

Usage Example: *Alice is seeking new TPLs for her Python project. Without PyRec, she explores a large number of TPLs hosted on PyPI and spends a long time reading the documents and testing the functionalities, interfaces, dependencies, and performance of each individual TPL. With PyRec, Alice chooses a few keywords that can sketch her project and lists the TPLs currently used or to be used in the project if any. Then, PyRec gives out a list of (say, 10) TPLs, which are potentially useful for her project. Now, Alice can focus on inspecting the usefulness of those recommended TPLs. Please note that PyRec is designed to recommend potentially useful TPLs for Alice to accelerate her TPL-seeking process. It is Alice who makes the final decisions. In addition, Alice can iteratively use PyRec to find new TPLs until she has successfully completed her project. Moreover, the trained KG can be used multiple times. It can also be easily retrained once projects or TPLs are updated. This helps PyRec include emerging TPLs and achieve even higher recommendation accuracy.*

3.2 Phase 1: Graph Generation

In this phase, PyRec generates the knowledge graph $\mathcal{KG}$ according to given TPL usage records (i.e., project-library interaction information) and contextual information.

Library Usage Records. These records represent the project-TPL interactions of all Python projects. To model

such information by graph, PyRec maps each Python project and TPL to an individual node in the bipartite graph $\mathcal{BG}$. Let us denote the set of Python projects as $\mathcal{P}$ and the set of TPLs as $\mathcal{L}$. Then, we have $\mathcal{BG} = (\mathcal{P}, \mathcal{R}_{\mathcal{PL}}, \mathcal{L})$, in which $\mathcal{R}_{\mathcal{PL}} = \{(p, r_0, l)|p \in \mathcal{P}, r_0 = 1, l \in \mathcal{L}\}$. A triple (p, r_0, l) represents the edge between project p and TPL l in $\mathcal{BG}$, i.e., the project-library interaction between p and l. Note that we use triple (p, r_0, l) rather than couple (p, l) here for the ease of combination with contextual information later.

Contextual Information. The contextual information is used to supplement the overlooked relationships between nodes in the $\mathcal{BG}$. It contains many real-world entities like developers, project categories, TPL categories, topics, etc. [32]. First, PyRec identifies those entities and creates the corresponding entity nodes in the $\mathcal{BG}$. For example, given two topic keywords *Education* and *Speech Recognition*, PyRec creates two nodes, one for each. Second, PyRec creates edges in $\mathcal{BG}$ between newly added entity nodes and existing nodes. It is worth noting that the dependencies between TPLs incur only new edges between the corresponding TPLs. Finally, the $\mathcal{BG}$ becomes a knowledge graph $\mathcal{KG}$.

As introduced in Section 2, each edge in $\mathcal{KG}$ has a specific type determined by the nodes it connects. We denote the set of newly added entity nodes that are related to projects as $\mathcal{E}_{\mathcal{P}}$, the set of newly added entity nodes that are related to TPLs as $\mathcal{E}_{\mathcal{L}}$, the set of newly added edges as $\mathcal{R}_{\mathcal{E}}$. Now, with contextual information, $\mathcal{KG}$ can be represented as $(\mathcal{H}, \mathcal{R}, \mathcal{T})$, in which the node sets $\mathcal{H}, \mathcal{T} \subset \mathcal{P} \cup \mathcal{E}_{\mathcal{P}} \cup \mathcal{E}_{\mathcal{L}}$, the edge set $\mathcal{R} = \mathcal{R}_{\mathcal{PL}} \cup \mathcal{R}_{\mathcal{E}}$ termed $\{(h, r, t)|h \in \mathcal{H}, r \in \mathcal{R}, t \in \mathcal{T}\}$. A triple (h, r, t) describes the relation r between head entity node h and tail entity node t. Please note that we treat those relations as bidirectional relations in this paper, e.g., l_2 *(torchaudio) is developed by* $author_1$ *(soumith)* and $author_1$ *(soumith) develops* l_2 *(torchaudio)* for a relation $(author_1, r_4, l_2)$.

3.3 Phase 2: Graph Embedding

Embedding has been widely used to learn latent features of entities in modern recommender systems [33]. Theoretically, the embedding process is to put similar entities, e.g., projects with similar functionalities, close to each other in the latent space [34]. PyRec implements a novel embedding mechanism as the information contained in the knowledge graph $\mathcal{KG}$ is much more complicated, i.e., there are different types of nodes and different types of edges in the graph. Given a triple $(h, r, t) \in \mathcal{KG}$, PyRec embeds nodes h and t into a d-dimensional *node space* [35]. The corresponding latent vectors are denoted as $e_h \in \mathbb{R}^d$ and $e_t \in \mathbb{R}^d$, respectively. The latent vector of a node can be interpreted as its features [15], [19]. For example, an embedding of TPL may represent its functionality, performance, popularity, compatibility, reliability, interface, etc. An embedding of a project may represent how much it is interested in each feature. Besides, PyRec embeds the relation r in triple (h, r, t) into a k-dimensional *relation space* [35]. The corresponding latent vector is denoted as $e_r \in \mathbb{R}^k$. The latent vector of a relation (edge) can be interpreted as its type, impact, importance, and usefulness of t to h, etc. Please note that d is not necessarily equal to k in practice.

PyRec employs the widely used TransR [35] to learn the latent vectors relevant to each triple (h, r, t) in $\mathcal{KG}$. Specifically, it projects e_h and e_t from the d-dimension node space to the k-dimension relation space. This can be done with the help of a trainable matrix $M_1 \in \mathbb{R}^{k \times d}$, i.e., $e_h^r = M_1 e_h$ where e_h^r is the projected embedding of h. Similarly, we have $e_t^r = M_1 e_t$. The learning process of e_r is to put the projected e_h^r and e_t^r close to e_r in the relation space of r. In other words, it tries to minimize the following equation.

$$f_r(h, t) = \|e_h^r + e_r - e_t^r\|_2^2 \tag{1}$$

where symbol $\|\cdot\|_2$ represents the Euclidean distance.

All embeddings are initialized with random values and learned during the training process (see Section 3.7).

3.4 Phase 3: Information Distillation

In this phase, for each node $h \in \mathcal{KG}$, e.g., Python project node p_1 and TPL node l_6 in Fig. 3, PyRec distills useful information from its neighbor nodes for subsequent TPL recommendation. This is done by GNN's message propagation mechanism which can capture information for a target node from its neighbor nodes in a graph [17], [36]. More importantly, different neighbor nodes may contribute different information of different levels of usefulness. Thus, PyRec applies attention mechanisms [34] to automatically adjust the importance of each neighbor node. We first discuss how to distill information from h's one-hop neighbor nodes, and then expand it to multiple hops.

Step 1: One-hop Information Distillation. PyRec employs $\mathcal{N}(h)$ to denote all relations in $\mathcal{KG}$ that take h as head node, i.e., $\mathcal{N}(h) = \{(h, r, t)|\exists(h, r, t) \in \mathcal{R}, \forall t \in \mathcal{T}\}$. Indeed, $\mathcal{N}(h)$ indicates the direct interactions between h and its one-hop neighbor nodes. Then, h's one-hop information, denoted as $e_{\mathcal{N}(h)}$, can be gathered as follows.

$$e_{\mathcal{N}(h)} = \sum_{\forall(h,r,t)\in\mathcal{N}(h)} w_r(h, t) e_t \tag{2}$$

Function $w_r(h, t)$ calculates the decay factor which controls how much information can be gathered from t along relation r. It is defined as follows.

$$w_r(h, t) = e_t^r \cdot f_{act}(e_h^r + e_r) \tag{3}$$

where symbol $(\cdot)$ denotes the *inner product*, $f_{act}()$ is the nonlinear activation function like *tanh* [34] used in this paper.

Remark: *PyRec applies an attention mechanism by including* $f_{act}(e_h^r + e_r)$ *in Eq. (3) to discriminate the importance of* h*'s neighbors [34], i.e, allowing node* t *to contribute more information to* h *if it is close to* h *in the relation space of* r*.*

Given all relations in $\mathcal{N}(h)$, PyRec adopts the Softmax function [33] shown below to normalize all decay factors.

$$w_r(h, t) = \frac{exp(w_r(h, t))}{\sum_{\forall(h,r*,t*)\in\mathcal{N}(h)} exp(w_{r*}(h, t*))} \tag{4}$$

Now, PyRec generates a vector with both the original embedding e_h and the information gathered from h's one-hop neighbors, i.e., $e_{\mathcal{N}(h)}$. We denote the vector as e_h^1.

$$\begin{aligned} e_h^1 =& LeakyReLU\Big(M_2\big(e_h + e_{\mathcal{N}(h)}\big)\Big) \\ &+ LeakyReLU\Big(M_3\big(e_h \odot e_{\mathcal{N}(h)}\big)\Big) \end{aligned} \tag{5}$$

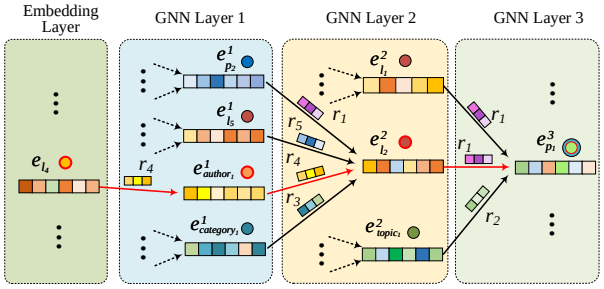

Fig. 4. Gathering multi-hop information from p_3 to for l_2

where $LeakyReLU()$ is the activation function [15], symbol $(\odot)$ is the *element-wise product*, and $M_2, M_3 \in \mathbb{R}^{d' \times d}$ are two trainable matrices used to transform e_h from the current GNN layer to the next GNN layer. d' is the transformation parameter. Its value is equal to the size of the next GNN layer.

Remark: *PyRec applies the second attention mechanism by including* $LeakyReLU\big(M_3(e_h \odot e_{\mathcal{N}(h)})\big)$ *in Eq. (5). It allows to selectively aggregate one-hop information, i.e., passing more information to* h *if* e_h *is closer to* $e_{\mathcal{N}(h)}$ *in latent space. We will experimentally study the effectiveness of the two attention mechanisms later in Section 4.5.*

Step 2: Multi-hop Information Distillation. PyRec stacks more GNN layers to capture the multi-hop information. Specifically, each GNN layer takes vectors produced by the previous layer as input and iterates the process introduced in Step 1 to generate new vectors. In this way, information possessed by neighbor nodes x-hops away from h in $\mathcal{KG}$ can be gathered by the x-th GNN layer. We iteratively define the embedding of h updated by the x-th layer as follows.

$$\begin{aligned} e_h^x = & LeakyReLU\left(M_2\left(e_h^{x-1} + e_{\mathcal{N}(h)}^{x-1}\right)\right) \\ & + LeakyReLU\left(M_3\left(e_h^{x-1} \odot e_{\mathcal{N}(h)}^{x-1}\right)\right) \end{aligned} \tag{6}$$

Example: ***Fig.*** *4 provides an example that* p_1 *distills 3-hop information from* l_4 *with 3 GNN layers, as* l_4 *connects to* p_1 *with 3 hops in Fig. 2 over path* $l_4 \xrightarrow{r_4} author_1 \xrightarrow{r_4} l_2 \xrightarrow{r_1} p_1$. *Latent vector* e_{l_4} *of node* l_4 *is initialized in Phase 2. Then, it is distilled through relation* r_4 *by the first GNN layer and merged to vector* $e^1_{author_1}$. *Next, it is distilled through relation* r_4 *by the second GNN layer and merged to vector* $e^2_{l_2}$. *Finally, it is merged into vector* $e^3_{p_1}$ *by the third GNN layer.*

3.5 Phase 4: Embedding Aggregation

In the previous phase, PyRec employs m-layer GNN to gather information for node h from its m-hop neighbor nodes in $\mathcal{KG}$. Each GNN layer produces an individual vector as output. In this phase, PyRec aggregates h's embedding and those generated vectors to constitute a final vector for h:

$$\overrightarrow{h} = e_h \| e_h^1 \| e_h^2 \| e_h^3 \| \cdots \| e_h^m \tag{7}$$

where $\|$ is the concatenation operation. Vector $\overrightarrow{h}$ possesses not only h's embedding but also useful information distilled from all its neighbor nodes within m hops.

3.6 Phase 5: TPL Prediction

As introduced in Section 3.3, the vector of a TPL node represents its features and the vector of a Python project node represents its interests in those features [15], [19]. Therefore, PyRec approximates the usefulness of TPL l to project p by:

$$\hat{u}(l,p) = \overrightarrow{l} \cdot \overrightarrow{p} \tag{8}$$

For each TPL $l \in L$, PyRec approximates its usefulness for p. Then, it recommends n TPLs with the highest usefulness values to developers of p. Upon the receipt of those TPLs, developers can prioritize the evaluation and find out if these recommended TPLs are indeed useful.

3.7 Optimization

Different from existing DL-based recommendation approaches [37], [38], PyRec optimizes the following two loss functions alternatively via Adam [39] to train the entire model, including graph embedding loss $\mathcal{L}_{rel}$ and TPL prediction loss $\mathcal{L}_{pre}$.

Graph Embedding Optimization. PyRec follows TransR [35] to optimize embeddings of $\mathcal{KG}$. Specifically, it considers both valid relations $\mathcal{R}$ and invalid relations $\mathcal{R}'$ in $\mathcal{KG}$ during the training. To generate $\mathcal{R}'$, it replaces node t in each valid triplet $(h,r,t) \in \mathcal{R}$ to a random node $t' \in \mathcal{T}$, such that $(h,r,t') \notin \mathcal{R}$. With $\mathcal{R}^* = \{(h,r,t,t') | (h,r,t) \in \mathcal{R}, (h,r,t') \in \mathcal{R}'\}$, PyRec minimizes the embedding loss:

$$\mathcal{L}_{rel} = \sum_{\forall (h,r,t,t') \in \mathcal{R}^*} -ln\sigma\Big(f_r(h,t') - f_r(h,t)\Big) \tag{9}$$

where $\sigma(\ )$ is the sigmoid function, function $f_r(\ )$ is calculated via Eq. (1). Eq. (9) indicates that PyRec tends to prioritize valid relations and penalize invalid relations.

Project-Library Interaction Optimization. Similarly, PyRec uses both valid project-library interactions $\mathcal{R}_{\mathcal{PL}}$ and invalid project-library interactions $\mathcal{R}_{\mathcal{PL}}{}'$ to optimize the TPL prediction. The generation of $\mathcal{R}_{\mathcal{PL}}{}'$ is the same as the generation of $\mathcal{R}'$ in the previous step. PyRec minimizes the following prediction loss:

$$\mathcal{L}_{pre} = \sum_{\forall (p,l,l') \in \mathcal{R}^*} -ln\sigma\Big(\hat{u}(l,p) - \hat{u}(l',p)\Big) \tag{10}$$

where $\mathcal{R}^* = \{(p,l,l') | (p,1,l) \in \mathcal{R}_{\mathcal{PL}}, (p,1,l') \in \mathcal{R}_{\mathcal{PL}}{}'\}$.

4 Experimental Evaluation

PyRec is designed to facilitate the project development for Python community. Specifically, it employs the DL-based mechanisms to automate developers' TPL seeking process. It is necessary to experimentally study the effectiveness of PyRec, i.e., if PyRec could perform better than state-of-the-art approaches. Second, considering that the scales of different Python projects in terms of TPL usage vary significantly, it is also of importance to explore the adaptability of PyRec to projects with different scales. Third, PyRec is the first approach that employs contextual information to make TPL recommendations. It also employs an attention mechanism to help automatically determine the importance of different kinds of information to the model. Thus, we conduct ablation studies to analyze the usefulness

of incorporating contextual information and the adoption of attention mechanism. After that, we want to study how to choose the most suitable parameters for PyRec in practice. Therefore, the following five research questions are used to guide the experimental evaluation of PyRec's effectiveness.

- **RQ1:** *Does PyRec perform better compared with existing state-of-the-art approaches?*
- **RQ2:** *Does PyRec perform well with Python projects of different scales?*
- **RQ3:** *Is contextual information useful for improving TPL recommendation performance?*
- **RQ4:** *Is attention mechanism useful for improving TPL recommendation performance?*
- **RQ5:** *How do PyRec's hyperparameter settings affect the recommendation performance?*

4.1 Experimental Setup

4.1.1 Dataset

Through a thorough investigation, we found that there is no benchmark dataset available, so we collected the dataset by adopting the following methodologies.

Project-TPL Usage Information Collection. We resort to the official GitHub API [40] to collect real-world Python projects. By setting the primary language to Python and excluding forked ones, we obtain around 13,000 projects. To collect TPLs for our research, we automatically retrieved TPLs from the official PyPI [4] repository. After removing the duplicates, we obtain around 6,000 TPLs for further analysis. Then, we leverage the static analysis framework Scalpel [23] to extract TPL usage information from collected Python projects. The advantage of Scalpel is that it can extract imported Python module names from the source code and exclude the standard modules and local modules. The top-level module names are used to extract used TPLs, similar to [41].

Contextual Information Collection. First, we retrieve the topic keywords of those Python projects when collecting them from GitHub. Those keywords are generated by GitHub and chosen by developers, and thus can accurately describe the corresponding projects. Second, we collect topic keywords, authors, and dependencies of each TPL from its installation wheel file. Please note that the above contextual information is publicly available and can be collected without knowing the usage status of a library. For example, once a library is available on PyPI, the corresponding contextual information like author name, description, dependencies, etc., can be easily collected and then used by PyRec.

Dataset Creation. Similar to [15], [19], we employ projects invoking 5 or more TPLs to conduct the evaluation. In total 121,474 project-library interaction records involving 12,421 projects and 963 distinct TPLs are used in the experiments. Besides, the dataset has 73,277 pieces of contextual records involving 9,675 extra entities like authors and keywords. As introduced in Section 3.1, the application scenario of PyRec is that developers have decided on a few TPLs for their Python projects and are seeking more new TPLs. Please note that from the programming perspective, there is no specific sequence for the TPL usage, i.e., if two TPLs are used by a project, developers can incorporate any of the two TPLs in the source code first, and then include the other one. The study of version evolution is out of the scope of this paper and will be studied in the future. In this paper, following the same experimental settings in [6], [15], [18], [19], we randomly remove rm TPLs in each Python project to mimic that some TPLs have been determined but some new TPLs are still needed. In addition, a project could be at different development stages. There is usually a limited number of TPLs used in a project at the early development stage, but more TPLs can be included when the development is nearly completed. To mimic such a real-world scenario, for each project in the dataset, we set $rm \in \{20\%, 40\%, 60\%\}$ TPLs. Here $rm = 60\% meansonly 40\% 4$ of TPLs have been determined and the developer wants to add 60% new TPLs (the removed ones in the experiments) to her/his project. This also indicates the project is at an early development stage. Similarly, rm=20% means the project is nearly completed and only 20% extra TPLs are needed. To generate the recommendation, we run PyRec to recommend a list with $n \in \{5, 10, 20\}$ TPLs. We investigate PyRec's performance by comparing those removed TPLs with those recommended TPLs. In this way, the removed TPLs constitute a test set and the remaining TPLs constitute a training set, the same as the settings in [6], [15], [18], [19]. Furthermore, as our PyRec uses contextual information to make recommendations, if a TPL is removed from a project, the relevant contextual information will also be removed from the training set accordingly. For ease of exposition, we call those TPLs kept in the test set as *correct TPLs* hereafter as the developers have used them eventually. The threats brought by the above settings will be discussed later in Section 4.7.

4.1.2 Implementation

We prototype PyRec in Python based on the state-of-the-art neural recommender system - KGAT [33]. However, we further improved KGAT to allow PyRec to utilize contextual information at both project and TPL sides. Besides, we improved KGAT to allow PyRec to support TPL dependencies which introduce new edges to $\mathcal{KG}$ without bringing new entities. For the other competing approaches [6], [15], [18], [19], we simply run their open-source codes with the Python dataset.

By default, we set the dimensionality of node embeddings $d = 128$, the dimensionality of relation embeddings $k = 64$, the number of GNN layers $m = 2$, and the size of each layer size $s = 64$ in PyRec. We adopt Adam [39] to adaptively adjust the learning rate. We keep the original parameters for the other competing approaches. The testbed is equipped with NVIDIA P100 12GB PCIe GPU accelerator. It runs Ubuntu 18.04, CUDA 10.2, Python 3.7.5, Torch 1.11.0, NumPy 1.21.5, pandas 1.3.5, SciPy 1.4.1, tqdm 4.64.0, and scikit-learn 0.22.

4.1.3 Metrics

Our objective is to propose an innovative approach to help Python developers effectively identify the most suitable TPLs. Therefore, we employ the first four metrics to evaluate PyRec's ability to recommend libraries accurately. Besides, we employ the last metric to measure PyRec's ability to

TABLE 1
Performance Comparison. Data with underlines are the best performance achieved by existing approaches.

Dataset	Approaches	n=5					n=10					n=20					Ave. Adv.
		MP	MR	MF	MRR	Cov	MP	MR	MF	MRR	Cov	MP	MR	MF	MRR	Cov	
rm=20%	LibRec	0.0819	0.2362	0.1216	0.2682	0.2669	0.0607	0.3731	0.0980	0.2850	0.2735	0.0332	0.3907	0.0583	0.3162	0.3089	39.51%
	CrossRec	0.0305	0.1136	0.0457	0.0723	0.1268	0.0262	0.1888	0.0442	0.0847	0.1458	0.0231	0.3333	0.0421	0.0964	0.1549	210.74%
	LibSeek	0.0985	0.3318	0.1418	0.2952	0.2847	0.0653	0.4291	0.1073	0.3092	0.3297	0.0413	0.5317	0.0739	0.3163	0.3765	18.60%
	GRec	0.0968	0.3394	0.1418	0.2926	0.3836	0.0645	0.4328	0.1074	0.3079	0.4593	0.0412	0.5390	0.0741	0.3153	0.5542	10.48%
	PyRec	**0.1106**	**0.3781**	**0.1711**	**0.3276**	**0.3857**	**0.0724**	**0.4844**	**0.1259**	**0.3424**	**0.4609**	**0.0405**	**0.5926**	**0.0845**	**0.3495**	**0.5553**	**69.83%**
rm=40%	LibRec	0.1497	0.2297	0.1672	0.3883	0.2778	0.0907	0.2616	0.1240	0.3932	0.2867	0.0497	0.2716	0.0781	0.3936	0.3186	64.60%
	CrossRec	0.0528	0.0912	0.0631	0.1279	0.1547	0.0522	0.1738	0.0759	0.1481	0.1965	0.0487	0.3286	0.0816	0.1651	0.2144	197.46%
	LibSeek	0.1675	0.2654	0.1907	0.4203	0.2979	0.1169	0.3598	0.1646	0.4359	0.3488	0.0774	0.4647	0.1256	0.4427	0.3915	29.59%
	GRec	0.1089	0.1928	0.1321	0.2680	0.4164	0.0804	0.2755	0.1184	0.2855	0.4968	0.0580	0.3798	0.0965	0.2951	0.5834	62.38%
	PyRec	**0.2111**	**0.3438**	**0.2616**	**0.5174**	**0.4215**	**0.1438**	**0.4549**	**0.2184**	**0.5312**	**0.4994**	**0.0926**	**0.5679**	**0.1591**	**0.5366**	**0.5852**	**88.51%**
rm=60%	LibRec	0.1246	0.1226	0.1138	0.3249	0.3279	0.0731	0.1353	0.0862	0.3268	0.3374	0.0396	0.1388	0.0561	0.3269	0.3443	158.08%
	CrossRec	0.0680	0.0738	0.0667	0.1290	0.2488	0.0880	0.1893	0.1133	0.1611	0.3032	0.0731	0.3245	0.1145	0.1768	0.3906	188.40%
	LibSeek	0.1705	0.1801	0.1643	0.3925	0.3507	0.1318	0.2715	0.1660	0.4094	0.4041	0.0887	0.3561	0.1340	0.4148	0.4219	59.23%
	GRec	0.1401	0.1578	0.1410	0.3178	0.5053	0.1039	0.2271	0.1350	0.3344	0.5721	0.0749	0.3154	0.1152	0.3432	0.6458	73.91%
	PyRec	**0.2862**	**0.3063**	**0.2959**	**0.6122**	**0.5331**	**0.2014**	**0.4196**	**0.2721**	**0.6229**	**0.6043**	**0.1320**	**0.5324**	**0.2116**	**0.6265**	**0.6781**	**119.91%**

recommend diverse TPLs. All metrics are widely used by researchers in not only the SE community but also the recommender system community. Greater values for each metric indicate better performance.

- **Mean Precision (MP)** [6], [15], [42]: Given a list consisting of n TPLs, the *precision* is calculated by dividing the number of correctly recommended TPLs by n. Then, MP averages all precisions in an experimental run.
- **Mean Recall (MR)** [6], [15], [16], [18]: The *recall* is calculated by dividing the number of correctly recommended TPLs in a list by the number of removed TPLs from the corresponding project. Then, MR averages the recalls of all lists in an experimental run.
- **Mean F1 Score (MF)** [15], [19]: MF averages the *F1-scores* of all lists in an experimental run. An F1-score is calculated with the precision and recall of a list.
- **Mean Reciprocal Rank (MRR)** [6], [19]: MRR measures the ability of each approach to put correct TPLs at higher positions in the recommendation list. Specifically, given a set of recommendation lists RL, the MRR is calculated by:

$$MRR = \frac{1}{|RL|} \sum_{\forall rl \in RL} \frac{1}{c(i)} \tag{11}$$

  where $c(i)$ is the position of the first correct TPL in the current recommendation list rl. Considering the fact that developers usually evaluate those recommended TPLs sequentially from top to bottom, a recommendation approach with higher MRR is much more useful in practice.
- **TPL Coverage (COV)** [6], [15], [19]. In one experimental run, COV is the ratio of distinct TPLs in all recommendation lists over the total number of distinct TPLs contained in the dataset. A greater value of COV indicates a higher probability that the approach recommends inventive TPLs. Note that inventive TPLs may not be correct TPLs and thus COV is irrelevant to accuracy. However, it can be used to check if an approach achieves better accuracy but significantly scarifies the diversity of recommended TPLs.

For each group of parameter settings, we conduct 50 experimental runs and report the averaged performance.

4.2 Performance Comparison (RQ1)

To answer the research question $RQ1$, we compare PyRec against four state-of-the-art approaches.

- **LibRec** [18]: It is the first TPL recommendation approach. It combines CF and association rule mining to recommend useful TPLs for Java projects.
- **CrossRec** [6]: It is a CF-based approach designed for open-source Java projects.
- **LibSeek** [19]: It is an MF-based TPL recommendation approach facilitating Android app development.
- **GRec** [15]: This is a DL-based approach designed for Android app development. It models the app-library interactions as a BG and employs GNN to distill information for TPL recommendations.

We simulate different development stages of Python projects by setting parameter rm to 20%, 40%, and 60%, respectively [6]. This means the developers have decided 80%, 60%, and 40%, respectively, of the TPLs for their Python projects and are seeking more TPLs for use. Then, given a Python project, each approach gives out a recommendation list consisting of n TPLs. In practice, the recommendation list could not be too long [15], [19], so we set n to $5, 10$, and 20, respectively. Table 1 reports the performance of all competing approaches. Please note that the *minimum advantages* (Min. Adv.) are calculated by comparing PyRec with the best performance achieved by state-of-the-arts (underlined). Besides, the *average advantages* (Ave. Adv.) are calculated by comparing PyRec with each of the competing approaches.

The first observation is that PyRec achieves the best performance in every case. On average across all cases, PyRec outperforms LibRec, CrossRec, LibSeek, and GRec by 90.67%, 181.30%, 31.18%, and 60.42% in MP, by 113.11%, 164.39%, 32.29%, and 52.65% in MR, and by 111.50%, 191.74%, 39.00%, and 66.04% in MF, respectively. This demonstrates PyRec's superior performance in recommending more correct TPLs for Python projects. Besides, PyRec outperforms LibRec, CrossRec, LibSeek, and GRec by 74.12%, 165.38%, 48.94%, and 3.69% in COV, respectively. This demonstrates that PyRec does not scarify the diversity of recommended TPLs while achieving higher accuracy than those state-of-the-art approaches. Moreover, compared

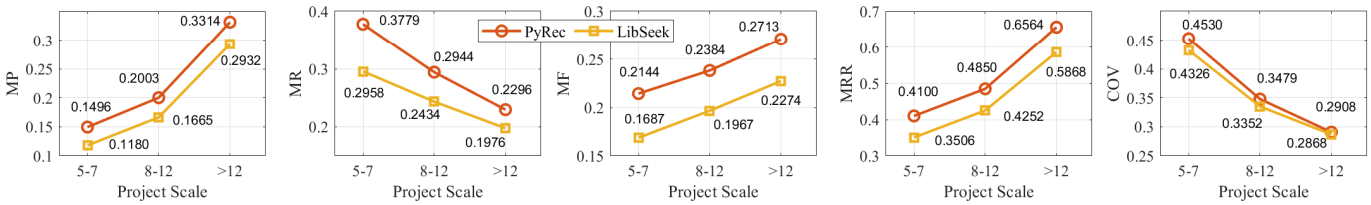

Fig. 5. Impact of Project Scales ($rm = 40\%, n = 5$)

with LibRec, CrossRec, LibSeek, and GRec, PyRec's average advantages in MRR are 47.58%, 291.52%, 28.62%, and 61.82%, respectively. This demonstrates PyRec's capability of putting those correct TPLs at higher positions in the recommendation lists. This is more helpful for developers as it helps prioritize the evaluation of useful TPLs and subsequently saves developers' TPL-seeking efforts.

The second observation is that along with the increase in n, the MR, MRR, and COV of PyRec increase accordingly. Taking $rm = 20\%$ as an example, when n increases from 5 to 20, the MR of PyRec increases from 0.3781 to 0.5926 by 56.72%, the MRR increases from 0.3276 to 0.3495 by 6.69%, and the COV increases from 0.3875 to 0.5553. When n is larger, more TPLs are included in each recommendation list. Therefore, PyRec has a higher probability to recommend not only correct TPLs but also inventive TPLs, which leads to an increase in MR, MRR, and COV. However, with a larger n, developers may spend more time testing those recommended TPLs. In practice, the more suitable value of n can be empirically set up according to developers' needs.

The third observation is that when rm increases, MP, MF, and MRR of PyRec increase accordingly. Given a Python project, a greater rm means that fewer TPLs have been decided by developers and more new TPLs are expected. Then, TPLs in a recommendation list have a higher probability to be the correct ones. This leads to an increase in MP, MF, and MRR. However, when more TPLs are expected, it is harder for PyRec to include all those correct TPLs in a list with a fixed length. As a result, its MR decreases slightly.

Interestingly, compared with nearly completed projects ($rm = 20\%$), PyRec achieves significant advantages when recommending TPLs for projects at earlier stages ($rm = 60\%$). For example, when $rm = 20\%$, PyRec's overall advantage is 69.83% on average across all cases. When rm increases to 60%, PyRec's overall advantage increases to 119.91%. The reason is that when rm increases, fewer TPLs are kept in the training set. Subsequently, less information can be utilized for recommendation by those approaches. This limits their performance. In contrast, PyRec is the only approach that can utilize contextual information to supplement TPL recommendations. This observation evidences our statement made in Section 1 that incorporating contextual information in TPL recommendation is useful.

The above observations demonstrate PyRec's suitability for Python projects not only nearing completion (e.g., $rm = 20\%$) but also at early development stage (e.g., $rm = 60\%$).

4.3 Adaptability to Projects Scales (RQ2)

Now we investigate PyRec's adaptability to projects with different scales. We split those Python projects into three categories according to the total number of TPLs used. The first category consists of projects using 5 to 7 TPLs, the second category consists of projects using 8-12 TPLs, and the last category consists of projects using more than 12 TPLs. Each category has a similar number of project-library interactions. As reported in **Table 1**, LibSeek has the best performance across all existing approaches. Thus, we employ LibSeek only for comparison. The results are shown in **Fig. 5**.

We can observe that the project scales significantly impact the performance of both PyRec and LibSeek. For example, in the first category where each project has 5 to 7 TPLs, PyRec achieves 0.1496 in MP. In contrast, in the third category where each project has more than 12 TPLs, PyRec's MP increases to 0.3314 by 221.47%. The reasons are two folds. First, as we remove 40% TPLs from each project as the test set, the theoretical up-bound of MP in the first category is 0.40 [19]. This up-bound is looser in the other two categories. Second, projects in the first category have relatively less TPL usage information. Compared with the second and third categories, it is harder to make accurate recommendations.

An interesting finding is that PyRec achieves the greatest advantage over LibSeek in the first category. Specifically, PyRec outperforms LibSeek by 24.65%, 19.14%, and 15.10% on average across the three categories. Because when a project invokes fewer TPls, less information can be utilized by LibSeek to make recommendations. This also demonstrates PyRec's ability to make accurate TPL recommendations with limited TPL usage information, as it can use contextual information as a supplement.

Another finding is that along with the increment of project scales, the COVs achieved by both approaches decrease accordingly. The reason is that when we fix $rm = 40\%$, a project with more TPLs will have more TPL usage information for the recommendation. Subsequently, each approach can make more accurate recommendations with fewer random TPLs included in the lists. As a result, the values of COV metric decrease. The last finding is that when the project scale becomes bigger, the advantage of PyRec over GRec in COV becomes smaller. This further evidences the observation reported in Table 1, i.e., PyRec is more useful for projects at early development stage.

4.4 Usefulness of Contextual Information (RQ3)

We conduct an ablation study to get deep insights into the effectiveness of utilizing contextual information. Indeed, the utilization of contextual information is one of the major differences between PyRec and GRec [15]. Specifically, we disable the attention mechanism (to avoid bias) defined by Eq.s (3) and (5). Then, we run PyRec without contextual information, denoted as PyRec$_{c0}$, and with contextual infor-

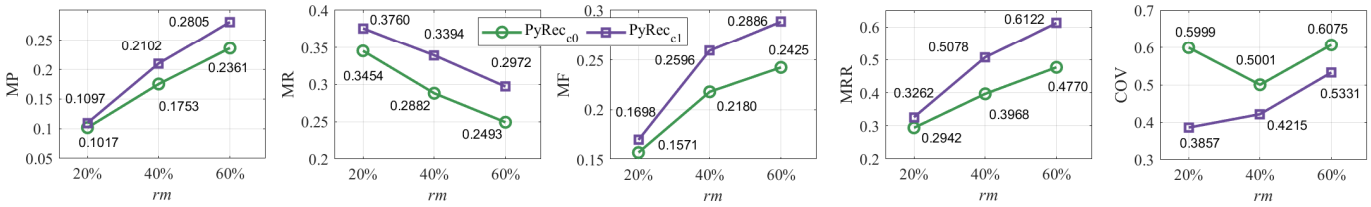

Fig. 6. Impact of Contextual Information ($n = 5$)

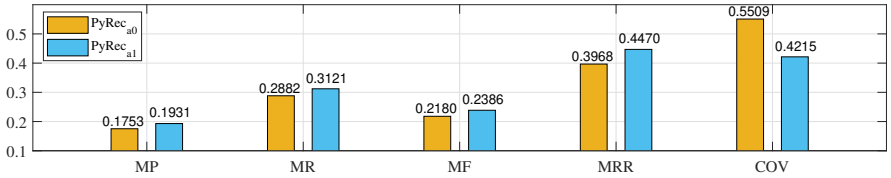

Fig. 7. Impact of Attention Mechanism ($rm = 40\%, n = 5$)

mation, denoted as PyRec$_{c1}$, separately. **Fig.** 6 depicts the final results when $n = 5$ and rm increases from 20% to 60%.

We can observe that the utilization of contextual information significantly boosts PyRec's performance in terms of recommendation accuracy. For example, PyRec$_{c1}$ outperforms PyRec$_{c0}$ by 8.92%, 21.17%, 21.34% on average when $rm = 20\%, 40\%$ and 60%, respectively. This evidences the statement made earlier in Section 1 that contextual information needs to be considered when recommending TPLs for Python projects. Second, the advantages of PyRec$_{c1}$ over PyRec$_{c0}$ increase when rm increases. This indicates that contextual information is much more useful for Python projects at the early development stage where a limited number of TPLs have been decided/used. This observation also confirms the findings shown in Table 1 that the advantage of PyRec over the other competing approaches becomes more significant when rm increases. Because all those approaches except PyRec used only project-library interaction information to make recommendations. When rm increases, less project-library interaction information is available for use, and their performance is highly constrained. In contrast, PyRec can use contextual information as a supplement when making recommendations, and thus it can have a much better performance.

Interestingly, the COV of PyRec is slightly lower than PyRec$_{c1}$. A potential reason is that when contextual information is incorporated into the model, PyRec could make more accurate TPL recommendations, and thus fewer fresh TPLs are included in the lists.

4.5 Usefulness of Attention Mechanism (RQ4)

Now we investigate whether the adoption of attention mechanisms in Section 3 (Eq.s (3) and (5)) can improve PyRec's performance. Similar to the settings in the previous section, we disable the usage of contextual information to avoid bias. Then, we change Eq. (3) to $w_r(h,t) = e_t^r$ and change Eq. (5) to $e_h^1 = LeakyReLU\big(M_2(e_h + e_{\mathcal{N}(h)})\big)$ to disable the attention mechanisms. We denote the new approach without attention mechanisms as PyRec$_{a0}$ and the approach with attention mechanisms as PyRec$_{a1}$. **Fig.** 7 shows the experimental results when $rm = 40\%$ and $n = 5$.

We can find that PyRec$_{a1}$ outperforms PyRec$_{a0}$ by 10.12%, 8.28%, 9.42%, and 12.64% in MP, MR, MF, and MRR, respectively. Because attention mechanisms can automatically formulate the importance of different neighbor nodes when gathering information for a target node in $\mathcal{KG}$. This helps amplify useful information possessed by neighbor nodes and filter out useless information. As a result, PyRec$_{a1}$ achieves much better performance. This observation evidences the effectiveness of attention mechanisms designed in Section 3. Similar to the phenomena observed in Fig. 6, PyRec$_{a0}$ performs better in COV than PyRec. The underlay reason is also the same.

4.6 Impact of PyRec's Hyperparameters (RQ5)

PyRec embeds both nodes (including Python project nodes, TPL nodes, and extra entity nodes) and edges (relations between nodes) in $\mathcal{KG}$ to latent space to capture their characteristics. Now we study the impact of different hyperparameters on PyRec's performance to answer research question RQ5.

Relation Embedding Dimensionality (k). We vary k to study its impact on PyRec's performance. Specifically, we set k to 16, 32, and 64 in PyRec$_{r1}$, PyRec$_{r2}$, and PyRec$_{r3}$, respectively. **Fig.** 8 reports the average performance achieved by each approach when $rm = 40\%$ and $n = 5$.

We can find that along with the increase of k, PyRec's recommendation accuracy becomes better in all cases. For example, compared with PyRec$_{r1}$ in which $k = 16$, PyRec$_{r2}$ improves the performance by 1.18%, 1.07%, 1.14%, and 0.57% in MP, MR, MF, and MRR, respectively. When $k = 64$, PyRec$_{r3}$ outperforms PyRec$_{r2}$ by 7.00%, 6.52%, 6.82%, and 9.03% in MP, MR, MF, and MRR, respectively. The reason is that a higher dimensionality of the relation embeddings allows PyRec to model more latent features for the corresponding relationships. Subsequently, more latent features allow PyRec to reflect the relations between each pair of nodes in $\mathcal{KG}$ more precisely. As a result, PyRec can recommend correct TPLs more effectively. In contrast, when k increases from 16 to 64, the value of COV decreases from 0.5395 to 0.4215 by 21.87%. Please note that a greater k also results in higher time consumption and more storage space. Thus, a proper value of k can be experimentally identified in practice.

Node Embedding Dimensionality (d). Now, we vary d to 32, 64, and 128, to study its impact on the performance of

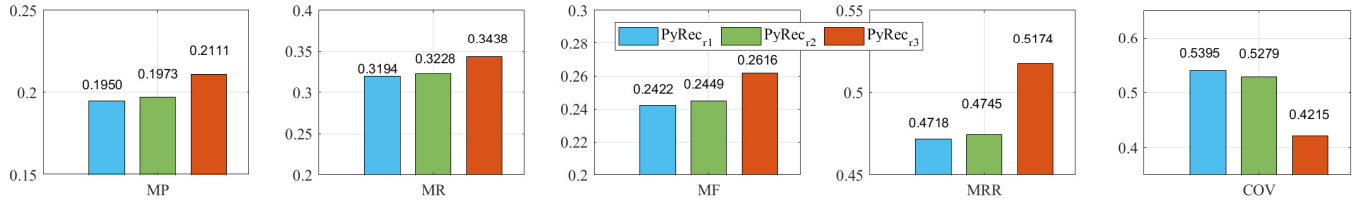

Fig. 8. Impact of Relation Embedding Size (k) ($rm = 40\%, n = 5$)

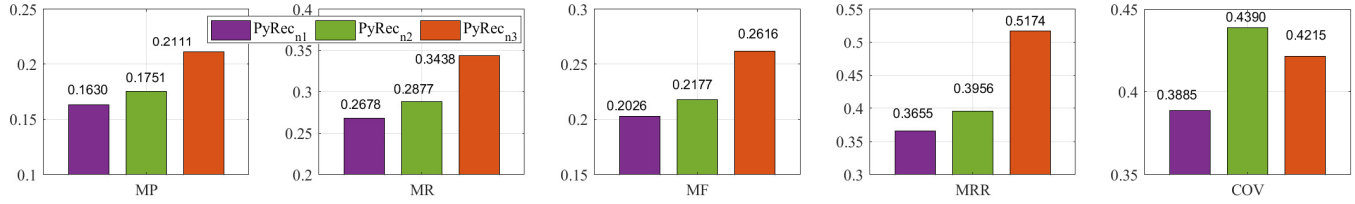

Fig. 9. Impact of Node Embedding Size (d) ($rm = 40\%, n = 5$)

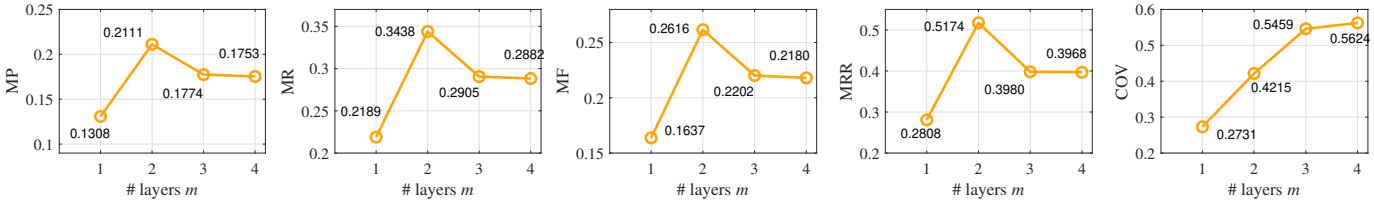

Fig. 10. Impact of Number of GNN Layers (m) ($rm = 40\%, n = 5$)

PyRec. We denote the three derived approaches as PyRec$_{n1}$, PyRec$_{n2}$, and PyRec$_{n3}$ in which d is 32, 64, and 128, respectively. The experimental results are reported in **Fig. 9**.

Similar to the phenomena observed before, a greater value of d results in better performance in MP, MR, MF, and MRR but COV. For example, PyRec$_{n2}$ outperforms PyRec$_{n1}$ by 7.45%, 7.43%, 7.44%, and 8.23% in MR, MR, MF, and MRR, respectively. Furthermore, compared with PyRec$_{n2}$, PyRec$_{n3}$'s advantages increase to 20.57%, 19.50%, 20.16%, and 30.78%, respectively. As introduced in Section 3.3, the embeddings of TPL nodes represent latent characters of those TPLs, such as functionality, performance, popularity, compatibility, reliability, interface, etc. The embeddings of Python project nodes represent how much they are interested in each feature. Therefore, a greater d allows a more accurate formulation of those characters and interests. Similarly, when d increases from 32 to 64, the value of COV increases accordingly. However, when d further increases from 64 to 128, the value of COV decreases slightly.

By comparing Fig.s 8 and 9, we can also observe that the node embedding dimensionality d has a more significant impact on PyRec's performance than the relation embedding dimensionality k.

Number of GNN Layers (m). As introduced in Section 3.4, PyRec uses the m-th GNN layer to capture information for target node from its m-hop neighbors in $\mathcal{KG}$. Now we vary m from 1 to 4 to study the impact of m on PyRec's performance, as shown in **Fig. 10**.

We can find that when m increases from 1 to 2, PyRec's recommendation accuracy increases rapidly, i.e., by 61.43%, 57.08%, 59.78%, and 84.27% in MP, MR, MF, and MRR, respectively. This evidences the effectiveness of capturing multi-hop information to facilitate the TPL recommendation, similar to [15]. Note that $m = 2$ is enough to capture all the contextual information as shown in Fig. 2. When m continues to increase, PyRec's performance decreases slightly, as an overly large m will include unnecessary noise that undermines GRec's accuracy. However, incorporating noisy information is beneficial for improving COV. Those fresh TPLs have a higher probability to be included in a recommendation list. As a result, the value of COV increases along with the increment of m.

Size of GNN Layers (s). Now we vary the GNN layer size s from 16 to 128 exponentially to study how it impacts PyRec's performance. As shown in Fig. 11, when s increases from 16 to 64, PyRec's recommendation accuracy increase rapidly. For example, the MP is 0.1835 when $s = 16$, and increases to 0.2111 by 15.06% when $s = 64$. This demonstrates that a greater s allows GNN to distill information more effectively. When s increases further, PyRec's performance decreases slightly, similar to the phenomena observed in Fig. 10. In contrast, the COV decreases when s increases from 16 to 64 but increases when s increases from 64 to 128. The reasons are similar and thus are omitted here. In practice, the optional s and m can be experimentally determined.

4.7 Threats to Validity

Internal Threats. The first threat comes from the dataset scale. Given the huge number of available projects and TPLs, we employed 12,421 projects and 963 distinct TPLs in the experiments, which may lead to bias. However, we collected 6,000 of the most popular TPLs for TPL usage analysis and randomly collected 13,000 Python projects published in the past 5 years. Therefore, the bias may exist but is not significant. The second thread comes from the implementation of PyRec and other competing approaches. To minimize this

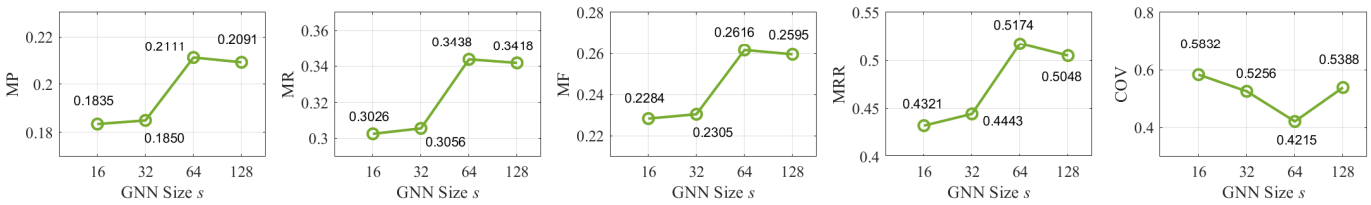

Fig. 11. Impact of GNN Layer Size (s) ($rm = 40\%, n = 5$)

thread, we made PyRec publicly available for the purpose of validation and reproduction of the experimental results. In addition, we used the original source codes and parameter settings of those competing approaches for comparison. The third threat comes from the correctness of the dataset. To mitigate this threat, we collected and manually filtered 13,000 Python projects from GitHub. Then, we employed a publicly available tool - Scalpel [23] - to extract the TPL usage information from each project. We also manually inspected the contextual information collected from GitHub and PyPI. Therefore, this threat has been minimized.

External validity. The main threat to the external validity comes from whether the PyRec proposed in this paper can be generalized to solve the TPL recommendation problems for applications developed in other programming languages. Although PyRec is a generalized TPL recommendation tool, we only conducted experimental evaluations with Python projects and Python TPLs. However, we employed four state-of-the-art approaches for comparison in the experiments. Those approaches were designed to solve the TPL recommendation problems for Java projects [18], open-source projects [6], and Android mobile app development [15], [19], respectively. The results reported in Table 1 show that PyRec has a significant advantage over those state-of-the-art approaches. We have also varied many parameters like rm and n to mimic different development scenarios to comprehensively evaluate PyRec's performance. Therefore, the threat exists but could not be significant. The second threat comes from that we did not repeat the user study to verify the usefulness of TPL recommendation for software development. However, preliminary studies [15], [19] have conducted comprehensive user studies and the usefulness of recommending TPLs for software development has been widely acknowledged by developers. We also take the two studies for comparison in the experiments. Therefore, the threat has been minimized.

Construct validity. The main threat comes from the four approaches used for comparison in the experiments. CrossRec [6] and LibSeek [19] can utilize the direct project-library interaction information. GRec can utilize higher-order interactions but cannot make use of contextual information. Therefore, their TPL recommendation performance tends to be lower than PyRec. To minimize this threat, we varied many parameters like rm, n, k, d, m and s to comprehensively evaluate PyRec's performance in different scenarios. Thus, this threat exists but is not significant. The second threat comes from the lack of project evolution information in the dataset. Almost all projects in the dataset are unique. If such evolution information is available, we can further investigate if a TPL recommended based on the current project will be used by its later versions. This can be used to supplement the experimental evaluation. However, as each project uses multiple TPLs at the same time, we followed the same settings in [6], [15], [18], [19] to conduct the experiments, i.e., randomly removing a specific portion of those TPLs and making recommendations based on the rest of TPLs. This simulates the practical development scenario where developers have determined part of TPLs and are seeking more TPLs for their projects. Indeed, the lack of evolution information will not affect the mechanism of PyRec. Therefore, the lack of project evolution information has a threat to the construct validity but will not be significant.

Conclusion validity. The first threat comes from the conclusion we made that PyRec can achieve high performance due to its ability to utilize both project-library interaction information and contextual information. The second threat comes from the conclusion we made that PyRec can achieve high performance due to the application of the attention mechanism in the GNN model. To minimize these two threats, we conducted a series of ablation studies by removing contextual information and/or attention mechanism, as shown in Section 4.4 and Section 4.5, respectively. This allows us to inspect PyRec's TPL recommendation performance with and without contextual information and attention mechanism. The last threat comes from the way we model the TPL usefulness. Following the same settings in [6], [15], [18], [19], we assume only TPLs in the test set are useful for the corresponding Python projects. However, TPLs beyond the test set may be also of interest. However, this will not scarify the performance reported in this paper. Therefore, this threat is not significant.

5 Related Work

In recent years, recommendation technique has been widely adopted to facilitate software development and evolution, such as defect identification [43], developer recommendation [44]–[47], API/code snippet recommendation [32], [48], third-party library recommendation [15], [19], permission recommendation [49], etc. Among them, our work is closely related to API recommendations and TPL recommendations. Besides, our work also is related to studies on Python libraries.

API Recommendation. Many approaches have been proposed to improve developers' coding efficiency by suggesting suitable APIs for use [50]–[53]. For example, given a request with feature descriptions, Thung et al. recommended APIs that can implement such features based on APIs' textual descriptions [50]. Huang et al. focused on mapping descriptions of developers' demands to APIs' structured descriptions when seeking useful APIs [51]. He et al. proposed a random forest-based approach to recommend

APIs based on data flow, token similarity, and token co-occurrence in Python [24]. Nguyen et al. employed a CF-based recommendation technique to find useful APIs for open-source projects [42]. Zhao et al. proposed APIMatchmaker to recommend APIs for Android app development [32]. To improve the recommendation accuracy, Liu et al. recommended APIs based on the API usage paths distilled from function call graphs [53]. Similarly, Xie et al. recommended new APIs by distilling hierarchical contextual information from the project's call graph [52]. Wu et al. proposed a neural framework leveraging multi-model fusion and multi-task learning techniques to recommend Web APIs [54]. Gong et al. [55] proposed DAWAR to improve the diversity and compatibility of recommended Web APIs. Different from the above work, PyRec recommends entire TPLs rather than specific program snippets or APIs.

TPL Recommendation. Many TPL usage patterns have been identified in recent years [11], [16], [56], which has created a foundation for TPL recommendation. Thung et al. made the first attempt at recommending TPLs for Java projects via combined association rule mining and CF [18]. Nguyen et al. employed solely CF to recommend TPLs for open-source Java software [6]. Later, He et al. proposed an MF-based approach to recommend TPLs for Android app development while diversifying the recommended TPLs [19]. Very recently, Li et al. employed GNN to recommend TPLs for Android app development based on the app-library graph [15]. However, all of the above approaches utilize only the TPL usage information to find useful TPLs. Different from them, our PyRec makes recommendations based on not only TPL usage information but also contextual information. It employs KG to model the heterogeneous relations between different entities and uses GNN to distill useful information from the graph. This takes a giant step out to advance TPL recommendation performance.

Python Library Studies. Recent studies on Python libraries have mainly focused on TPL/API evolution [2], [3], [22] and TPL dependency analysis [57]–[60]. To name a few, He et al. studied the TPL migration problem and proposed a novel approach that utilizes TPL characteristics like rule support, message support, distance support, and API support to rank TPLs and recommend migration solutions [61]. Zhang et al. investigated the API evolution in Python libraries and detected compatibility issues caused by such API evolution [3]. They proposed PYCOMPAT to automatically detect compatibility issues caused by the misuse of evolved APIs. Similarly, Wang et al. investigated how the deprecated APIs are declared in Python libraries and handled in Python projects [2]. Rubei et al. [22] integrated end-user feedback to recommend developers with TPL evolution suggestions, i.e., whether to keep or discard a used TPL. Cheng et al. modeled relations of TPLs into a KG to infer compatible runtime environments [41]. Wang et al. studied the TPL dependency issues in Jupyter Notebooks [58]. They presented SnifferDog to automatically restore the execution environment for Jupyter Notebooks based on TPL usage analysis. Ye et al. proposed PyEGo to automatically infer dependencies between not only TPLs but also Python interpreter and system libraries [59]. Ying et al. proposed Watchman to automatically detect dependency conflicts among Python TPLs for the PyPI ecosystem [60]. The above studies on TPL dependency supplement PyRec with plentiful contextual information.

6 Conclusion and Future Work

In this paper, we proposed innovative PyRec to facilitate the development of software projects. PyRec helps relieve developers' burden incurred by seeking new TPLs for their projects. Unlike previous approaches that employ solely existing TPL usage information to make recommendations, PyRec leverages both TPL usage information and contextual information by encoding them into a knowledge graph. This enables PyRec to gather more information via GNN to make more accurate recommendations. More domain-specific techniques like attention mechanism are also incorporated in PyRec to further burst its performance. The experimental results on 12,421 Python projects demonstrate the superior performance of PyRec.

In the future, we will explore the possibility of recommending TPL updates for software applications, e.g., giving TPL update suggestions and/or TPL migration suggestions.

References

[1] "Top programming languages 2021," https://spectrum.ieee.org/top-programming-languages/, accessed: 2023-05-30.

[2] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 233–244.

[3] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do Python framework APIs evolve? An exploratory study," in *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 81–92.

[4] "PyPI - the Python package index," https://pypi.org/, accessed: 2023-05-30.

[5] B. Xu, L. An, F. Thung, F. Khomh, and D. Lo, "Why reinventing the wheels? An empirical study on library reuse and re-implementation," *Empirical Software Engineering*, vol. 25, no. 1, pp. 755–789, 2020.

[6] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "CrossRec: Supporting software developers by recommending third-party libraries," *Journal of Systems and Software*, vol. 161, p. 110460, 2020.

[7] 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.

[8] 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.

[9] 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.

[10] 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 (ICSE)*. ACM, 2016, pp. 653–656.

[11] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[12] M. Lamothe and W. Shang, "When APIs are intentionally bypassed: An exploratory study of API workarounds," in *42nd International Conference on Software Engineering (ICSE)*, vol. 2020, 2020.

[13] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "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)*, 2020.

[14] T. Ki, C. M. Park, K. Dantu, S. Y. Ko, and L. Ziarek, "Mimic: UI compatibility testing system for Android apps," in *41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 246–256.

[15] B. Li, Q. He, F. Chen, X. Xia, L. Li, J. Grundy, and Y. Yang, "Embedding app-library graph for neural third party library recommendation," in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 466–477.

[16] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *Journal of Systems and Software*, vol. 145, pp. 164–179, 2018.

[17] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–38, 2019.

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

[19] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang, "Diversified third-party library prediction for mobile app development," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 150–165, 2022.

[20] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin, "Incorporating contextual information in recommender systems using a multidimensional approach," *ACM Transactions on Information Systems*, vol. 23, no. 1, pp. 103–145, 2005.

[21] "Topics on GitHub," https://github.com/topics, accessed: 2023-05-30.

[22] R. Rubei, C. Di Sipio, J. Di Rocco, D. Di Ruscio, and P. T. Nguyen, "Endowing third-party libraries recommender systems with explicit user feedback mechanisms," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 817–821.

[23] L. Li, J. Wang, and H. Quan, "Scalpel: The Python static analysis framework," *arXiv preprint arXiv:2202.11840*, 2022.

[24] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, and B. Xu, "PyART: Python API recommendation in real-time," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1634–1645.

[25] "torchaudio: An audio library for PyTorch," https://github.com/pytorch/audio, accessed: 2023-05-30.

[26] "Matplotlib: Visualization with Python," https://matplotlib.org/, accessed: 2023-05-30.

[27] "PyYAML," https://pyyaml.org/, accessed: 2023-05-30.

[28] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM computing surveys (CSUR)*, vol. 52, no. 1, pp. 1–38, 2019.

[29] "PyTorchVideo: A deep learning library for video understanding research," https://pytorchvideo.org/, accessed: 2023-05-30.

[30] "scikit-sound: Python utilites for working with sound signals," https://pypi.org/project/scikit-sound/, accessed: 2023-05-30.

[31] "SQLAlchemy: The database toolkit for Python," https://www.sqlalchemy.org/, accessed: 2023-05-30.

[32] Y. Zhao, L. Li, H. Wang, Q. He, and J. Grundy, "APIMatchmaker: Matching the right APIs for supporting the development of Android apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 113–130, 2022.

[33] X. Wang, X. He, Y. Cao, M. Liu, and T.-S. Chua, "KGAT: Knowledge graph attention network for recommendation," in *25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 950–958.

[34] Z. Li, H. Liu, Z. Zhang, T. Liu, and N. N. Xiong, "Learning knowledge graph embedding with heterogeneous relation attention networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 8, pp. 3961–3973, 2021.

[35] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion," in *29th AAAI Conference on Artificial Intelligence*, 2015.

[36] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[37] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *26th International Conference on World Wide Web (WWW)*, 2017, pp. 173–182.

[38] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, "Neural graph collaborative filtering," in *42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2019, pp. 165–174.

[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Poster of International Conference on Learning Representations (ICLR)*, 2015.

[40] "Search - GitHub docs," https://docs.github.com/en/rest/search#search-repositories, accessed: 2023-05-30.

[41] W. Cheng, X. Zhu, and W. Hu, "Conflict-aware inference of Python compatible runtime environments with domain knowledge graph," in *44th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 2022, p. 451–461.

[42] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining API function calls and usage patterns," in *41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.

[43] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "ReCDroid: Automatically reproducing Android application crashes from bug reports," in *41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 128–139.

[44] L. Ye, H. Sun, X. Wang, and J. Wang, "Personalized teammate recommendation for crowdsourced software developers," in *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 808–813.

[45] L. Bao, X. Xia, D. Lo, and G. C. Murphy, "A large scale study of long-time contributor prediction for github projects," *IEEE Transactions on Software Engineering*, 2019.

[46] D. Kong, Q. Chen, L. Bao, C. Sun, X. Xia, and S. Li, "Recommending code reviewers for proprietary software projects: A large scale study," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 630–640.

[47] X. Xie, X. Yang, B. Wang, and Q. He, "DevRec: Multi-relationship embedded software developer recommendation," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4357–4379, 2021.

[48] Y. Zhao, L. Li, X. Sun, P. Liu, and J. Grundy, "Icon2Code: Recommending code implementations for Android GUI components," *Information and Software Technology (IST)*, 2021.

[49] Z. Liu, X. Xia, D. Lo, and J. Grundy, "Automatic, highly accurate app permission recommendation," *Automated Software Engineering*, vol. 26, no. 2, pp. 241–274, 2019.

[50] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 290–300.

[51] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-API knowledge gap," in *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 293–304.

[52] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "HiRec: API recommendation using hierarchical context," in *30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 369–379.

[53] X. Liu, L. Huang, and V. Ng, "Effective API recommendation without historical software repositories," in *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 282–292.

[54] H. Wu, Y. Duan, K. Yue, and L. Zhang, "Mashup-oriented Web API recommendation via multi-model fusion and multi-task learning," *IEEE Transactions on Services Computing*, vol. 15, no. 6, pp. 3330–3343, 2021.

[55] W. Gong, X. Zhang, Y. Chen, Q. He, A. Beheshti, X. Xu, C. Yan, and L. Qi, "DAWAR: Diversity-aware Web APIs recommendation for mashup creation based on correlation graph," in *45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2022, pp. 395–404.

[56] M. A. Saied and H. Sahraoui, "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, 2016, pp. 1–10.

[57] E. Horton and C. Parnin, "DockerizeMe: Automatic inference of environment dependencies for Python code snippets," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 328–338.

[58] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of Jupyter notebooks," in *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1622–1633.

[59] H. Ye, W. Chen, W. Dou, G. Wu, and J. Wei, "Knowledge-based environment dependency inference for Python programs," in *IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1245–1256.

[60] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for python library ecosystem," in *ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 125–135.

[61] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 72–83.