Neural Library Recommendation by Embedding Project-Library Knowledge Graph

Bo Li, Haowei Quan, Jiawei Wang, Pei Liu, Haipeng Cai, Yuan Miao, 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 significantly in all cases.**

***Index Terms*—Third-party library, recommendation, knowledge graph, graph neural network, Python.**

I. 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 [1]. One reason that fuels Python's popularity could be the large number of third-party libraries (TPLs) readily to be used by the community [2]. 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[2].

Compared with programming from scratch, TPLs offer tailor-made APIs with the same functionalities [3] but less bugs/deficiencies [4]. Therefore, seeking TPLs with desired functionalities and integrating them in projects under development is much more effective [5], [6], [7], [8]. Indeed, it has become a common practice for developers to regularly use TPLs to accelerate their development process and/or deliver new features [9].

Unfortunately, given the huge number of TPLs available for use, it is challenging for developers to seek the most suitable TPLs for their projects [10], [11]. First, manually inspecting the functionalities, interfaces, performance, etc., of tremendous TPLs is very time-consuming [12]. It is even more sophisticated nowadays as TPLs are evolving rapidly [1] and the time-to-market constraint is becoming tighter [13]. Second, TPL usage has specific characteristics [9], [14], e.g., combinations and dependencies. Finding appropriate TPLs fulfilling such characteristics is another time-consuming process [3], [4].

Inspired by the great success of recommender systems in a variety of domains [15], many TPL recommendation approaches have been proposed recently to accelerate the TPL seeking process [4], [13], [16], [17]. Generally, they provide developers with a short list of TPLs for consideration. For example, LibRec [16] and CrossRec [4] 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 [17] 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 [13] 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 [13], [17]. However, the performance of existing

Manuscript received 24 June 2023; revised 7 April 2024; accepted 22 April 2024. Date of publication 24 April 2024; date of current version 14 June 2024. Recommended for acceptance by S. Nejati. *(Corresponding author: Li Li.)*

Bo Li and Yuan Miao are with the College of Arts, Business, Law Education, and Information Technology, Victoria University, Melbourne, VIC 3011, Australia (e-mail: bo.li@vu.edu.au; yuan.miao@vu.edu.au).

Haowei Quan, Jiawei Wang, and Pei Liu are with the Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia (e-mail: haowei.quan@monash.edu; jiawei.wang1@monash.edu; pei.liu@monash.edu).

Haipeng Cai is with the School of Electrical Engineering and Computer Science (EECS), Washington State University, Pullman, WA 99163 USA (e-mail: haipeng.cai@wsu.edu).

Yun Yang is with the Department of Computing Technologies, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: yyang@swin.edu.au).

Li Li is with the School of Software, Beihang University, Beijing 100191 China (e-mail: lilicoding@ieee.org).

Digital Object Identifier 10.1109/TSE.2024.3393504

[1]https://spectrum.ieee.org/top-programming-languages/

[2]https://pypi.org/

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 recommendations. 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 [18] like the inherent relations between different projects and different TPLs, which have been overlooked by existing approaches. For example, projects with the same topic[3] 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 [13]. 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, is helpful in finding more suitable TPLs [19]. However, existing approaches have unfortunately ignored such information and thus their performance is constrained.

In this paper, we take the TPL recommendation for Python projects as a use case of our approach. The reason is that Python has emerged as the most popular programming language in recent years. There is already a huge number of Python projects and the amount of Python libraries available for use [20]. In addition, Python developers rely on the rich functionalities offered by the huge collection of TPLs for fast prototyping [1], [20], [21]. 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 our approach is language agnostic. It can be applied to projects developed in any programming language, such as mobile apps and conventional Java projects, wherever the required contextual information is available. The adaptation to other languages will be explored in the future.

PyRec is an innovative approach that makes full use of both project-library interaction information and available 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 [16], CrossRec [4], LibSeek [17], and GRec [13], PyRec can make more accurate recommendations. The key contributions of this research are concluded as follows.

[3]https://github.com/topics

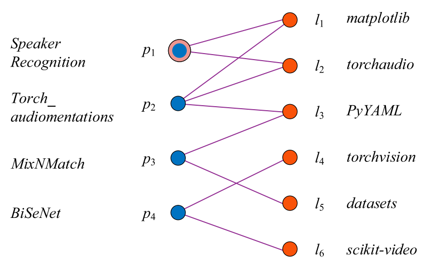

Fig. 1. Exemplar project-library bipartite graph $\mathcal{BG}$.

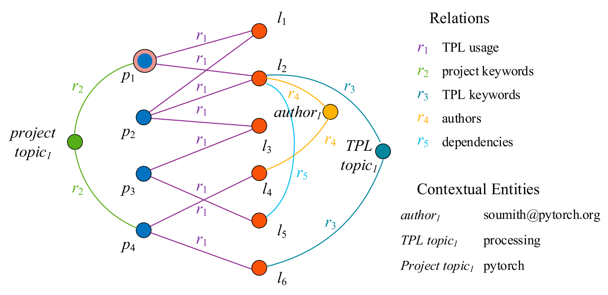

Fig. 2. Exemplar project-library knowledge graph $\mathcal{KG}$.

- We are the first TPL recommendation study that utilizes both project-library interaction information and contextual information.
- We model Python projects, TPLs, project-TPL interactions, relations between projects, and relations between TPLs as a KG, in which comprehensive relations between different projects and different TPLs are represented. This allows capturing more information crucial for accurate TPL recommendation.
- Inspired by KGAT [22], we propose an innovative GNN-based DL model to distill useful information from the generated KG for TPL recommendation. While KGAT is initially designed for commerce recommendation, this paper is the first attempt at recommending TPLs based on a knowledge graph. Besides, KGAT can only create relations between TPLs based on contextual information. However, our model can model the relations between different projects and different TPLs. This allows better utilization of contextual information. 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 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. Both PyRec and our dataset are publicly available[4] for validation and reproduction of our experimental results.

[4]https://github.com/Limber0117/PyRec

The rest of this paper is organized as follows. Section II motivates the research of this paper. Section III introduces PyRec in detail. Section IV evaluates PyRec experimentally. Section V reviews related work. Then, Section VII concludes this paper and points out future work.

II. Motivating Example

Fig. 1 provides an exemplar bipartite graph (denoted as $\mathcal{BG}$) modeling the project-library interactions. Specifically, it has 4 project nodes including p_1 (*SpeakerRecognition*[5]), p_2 (*Torch_audiomentations*[6]), p_3 (*MixNMatch*[7]), and p_4 (*BiSeNet*[8]). Those projects invoke 6 TPLs, including *matplotlib*, *torchaudio*, *PyYAML*, *torchvision*, *datasets*, and *scikit-video*, 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*) indicates that Python project p_1 uses TPL *torchaudio*. In the following discussions, we use different approaches to recommend new TPLs for project p_1.

CF-based approaches like CrossRec [4] 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*) and l_2 (*torchaudio*). In the meantime, project p_2 also invokes these two TPLs. Based on such TPL usage information, p_1 and p_2 have a similarity of $2/3$. Thus, TPLs used in p_2 but not yet in p_1 will be recommended to p_1, i.e., CrossRec will recommend l_3 (*PyYAML*) 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 [17] 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. After the embedding, LibSeek calculates dot products for p_1's embedding and embeddings of all other TPLs that have not been used by p_1, i.e., embeddings of l_3, l_4, l_5 and l_6. Then, it recommends the TPL with the largest dot product value, e.g., l_3 (*PyYAML*) in this example.

In recent years, deep learning has been widely adopted to solve a variety of technical problems. Many DL-based recommender systems were proposed by the software engineering (SE) community. GRec [13] is the state-of-the-art DL-based approach that is capable of exploiting transitive information in $\mathcal{BG}$ to learn the latent vectors. For example, in $\mathcal{BG}$ shown in Fig. 1, there is a path l_3-p_2-$l_2(l_1)$-p_1 connecting p_1 and l_3. Similarly, there is a path l_5-p_3-l_3-p_2-$l_2(l_1)$-p_1 connecting l_5 and p_1. Therefore, GRec recommends both l_3 (*PyYAML*) and l_5 (*datasets*) to p_1. The utilization of transitive information boosts GRec's TPL recommendation performance.

However, the $\mathcal{BG}$ used by the above-mentioned approaches cannot reflect all the relationships between those projects and TPLs. Therefore, based on $\mathcal{BG}$, 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 $project\ topic_1$ (*pytorch*) on GitHub, and thus have similarities. 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 (*torchvision*) and l_6 (*scikit-sound*).

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 between p_1 and l_1 represents the TPL usage interactions (denoted as r_1), the edge between l_2 and node of entity $TPL\ topic_1$ (*processing*) indicates that *torchaudio* belongs to category *media processor* on PyPI (denoted as r_3), etc. Dependency is a unique characteristic of TPLs. To model such information, we add new edges between each pair of involved TPLs in $\mathcal{BG}$. For example, l_5 (*datasets*) has a dependency relationship with l_2 (*torchaudio*), we have an edge between l_5 and l_3 with the type of r_5.

With $\mathcal{KG}$, we can recommend more TPLs beneficial for p_1. For example, there is a new path $l_4(torchvision)$-$author_1$-$l_2(torchaudio)$-p_1 between l_4 and p_1. Then, in addition to l_3 and l_5 that have been recommended by GRec, l_4 can further be recommended to p_1. Such a recommendation is useful in practice because p_1 is a speaker recognition project that extracts the speaker's features based on ResNet [23] - a famous computer vision model. Therefore, both voice processing and image/video processing functionalities are important for p_1. Here, l_2 (*torchaudio*) and l_4 (*torchvision*) fulfill this requirement. Moreover, both l_2 and l_5 are proposed by the same developer identified by the email *soumith@pytorch.org*. They usually have similar conventions, coding styles, design patterns, and shared APIs. Compared with other TPLs providing similar functionalities to l_5 $(torchvision)$, developers can integrate $torchvision$ into their mobile apps more easily. The second example is the recommendation of l_5 (*datasets*). Although GRec can also recommend l_5, using $\mathcal{KG}$ can further prioritize l_5 in the recommendation list, because an extra path l_5-l_2-p_1 is created between l_5 and p_1 in $\mathcal{KG}$. This is meaningful in practice. Considering that l_5 depends on l_2, it usually provides extra

[5]https://github.com/jymsuper/SpeakerRecognition_tutorial, #936 in the dataset introduced later in Section IV-A1

[6]https://github.com/asteroid-team/torch-audiomentations, #1063

[7]https://github.com/Yuheng-Li/MixNMatch, #4416

[8]https://github.com/CoinCheung/BiSeNet, #11962

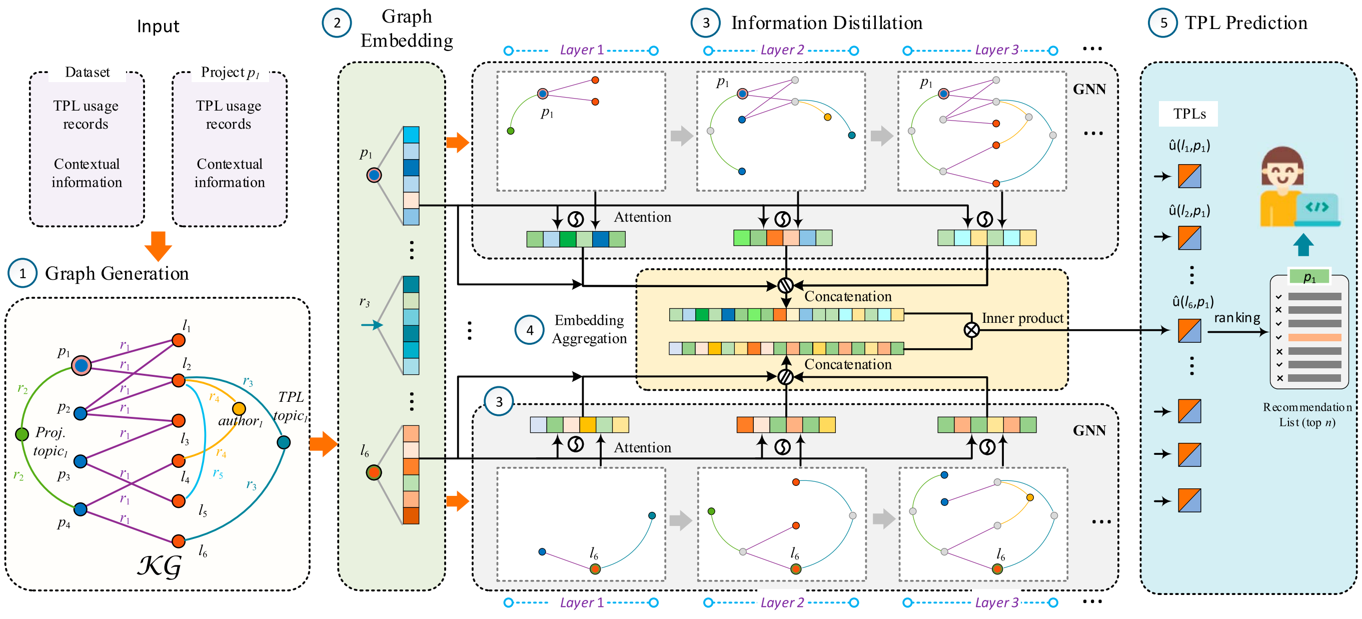

Fig. 3. General process flow of PyRec.

features than l_2, and thus the inclusion of l_5 could be beneficial for p_1. Based on the above examples we can find that extending $\mathcal{GB}$ to $\mathcal{KG}$ with contextual information is practical and useful.

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 have the same topic keyword in PyPi, and l_5 connects to 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.

III. PyRec Approach

A. 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 [13] that has only project nodes and TPL nodes, PyRec identifies new entity nodes, e.g., $project\ topic_1$, $author_1$, and $TPL\ topic_1$ 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 [13] 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 the top n useful TPLs that have not been used by p_1. Different from GRec [13], 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, that 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. Alice can iteratively use the trained PyRec to find new TPLs from $\mathcal{KG}$ until she completes her project.*

The same as modern GNN-based recommender systems like [13], [24], [25], PyRec can be periodically retrained [26] to incorporate new TPLs, contextual information, and projects. The overall process can take several minutes to a few hours, depending on the scale of the dataset. For example, given a graph with 13,000 nodes and 200,000 edges, it takes less than two hours (Section IV-G) on our testbed (Section IV-A2). Firstly, the training dataset is updated to include the new TPLs, projects, and contextual information. Secondly, the $\mathcal{KG}$ is generated according to the methods provided later in Section III-B. Thirdly, PyRec is trained based on the updated graph. The retraining helps PyRec include emerging TPLs and achieve even higher recommendation accuracy.

The design of PyRec is inspired by the state-of-the-art neural recommender system - KGAT [22]. However, there are fundamental differences between KGAT and PyRec. **Firstly**, KGAT is a recommender designed to recommend e-commerce products. It utilizes contextual information about products to model the relationship between those products. In contrast, PyRec not only models the relationship between projects but also models the relationship between TPLs. This allows PyRec to better utilize the contextual information. **Secondly**, KGAT forms the relationships between products by adding new entities extracted from contextual information into $\mathcal{KG}$. However, the relationships between TPLs are more complicated. For example, two TPLs may have the same author(s). Then, an author entity is created in the knowledge graph to establish the relationship between the two TPLs, similar to KGAT. However, a TPL may be designed based on another TPL. Such a dependency relationship does not involve any new entity in $\mathcal{KG}$. Instead, there is an edge in $\mathcal{KG}$ between the two TPLs. Therefore, PyRec has unique design to accommodate such TPL feature. **Thirdly**, as more complicated relationships are included in $\mathcal{KG}$, it is essential to automatically determine the usefulness of each piece of information distilled from $\mathcal{KG}$. Therefore, PyRec has a dedicated attention mechanism to enable the usefulness identification for different neighbor nodes and relations. Based on it, PyRec can distill more useful information and mitigate the negative impact of unuseful information.

B. 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 $\mathcal{BG}$. Theoretically, it can include any real-world information like developers, project categories, TPL categories, topics, etc. [27]. First, PyRec identifies new entities from contextual information 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 II, 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 $(author_1, r_4, l_2)$.

C. Phase 2: Graph Embedding

Graph Embedding has been well-studied in the past years. Many techniques like TransH, TransE, TransD, DistMult, and ComplEx were proposed to learn the latent features of entities in a given knowledge graph [28]. The embedding process is to learn the latent vectors for nodes and edges in $\mathcal{KG}$, putting similar entities, e.g., projects with similar functionalities, close to each other in the latent space [29]. Please note that theoretically any graph embedding technique can be employed by PyRec to embed $\mathcal{KG}$. However, TransR [30] has a better performance when handling complex knowledge graphs consisting of a large scale of nodes and various kinds of edges [31]. Besides, TransR has been widely used to implement knowledge graph-based recommender systems [32]. Considering its popularity, maturity, and fitness, PyRec employs TransR to implement the graph embedding, the same as KGAT [22]. We will study the impact of different embedding techniques on PyRec's performance in the future.

There are different types of nodes and different types of edges in the graph $\mathcal{KG}$. Given a triple $(h, r, t) \in \mathcal{KG}$, PyRec embeds nodes h and t into a d-dimensional *node space* [30]. 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 [13], [17]. 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* [30]. 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 [30] 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 III-G).

D. 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 [15], [33]. More importantly, different neighbor nodes may contribute different information of different levels of usefulness. Thus, PyRec applies attention mechanisms [29] 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.

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

where symbol $(\cdot)$ denotes the *inner product*, $f_{act}()$ is the nonlinear activation function like *tanh* [29] 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 [29], 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 [22] 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.

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

where $LeakyReLU()$ is the activation function [13], 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 IV-E.*

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.

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

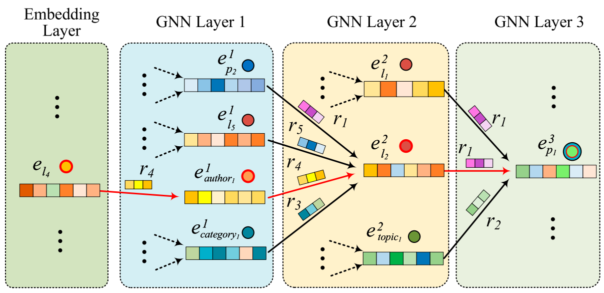

Fig. 4. Gathering multi-hop information from p_3 to for l_2.

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.*

E. 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:

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

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.

F. Phase 5: TPL Prediction

As introduced in Section III-C, the vector of a TPL node represents its features and the vector of a Python project node represents its interests in those features [13], [17]. 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 in total that have the highest usefulness values but have not been used by p 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.

G. Optimization

Different from existing DL-based recommendation approaches [34], [35], PyRec optimizes the following two loss functions alternatively via Adam [36] 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 [30] 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 $\mathcal{R_{PL}}$ and invalid project-library interactions $\mathcal{R_{PL}}'$ to optimize the TPL prediction. The generation of $\mathcal{R_{PL}}'$ 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 $\mathcal{R}^* = \{(p,l,l') | (p,1,l) \in \mathcal{R_{PL}}, (p,1,l') \in \mathcal{R_{PL}}'\}$.

IV. 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?*

A. Experimental Setup

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[9] to collect the most popular Python projects. By setting the primary language to Python and excluding forked ones, we obtain around 13,000 projects. To collect potential TPLs used in those projects, we retrieve 6,000 of the most popular libraries from the official PyPI[10] repository. Please note that those library names are used as ground truth to verify the TPLs extracted from each project. We leverage the static analysis framework Scalpel [20] to extract TPL usage information from collected Python projects. Specifically, given a project, Scalpel extracts unique Python module names by inspecting the import statements in each source file. Then, it excludes the standard modules and local modules. When submodules are used in a given source file, Scalpel only collects the top-level module names, similar to [37]. Then, the extracted TPLs rae compared against the library names collected from PyPI to ensure their correctness.

Contextual Information Collection. A variety of information can be used as contextual information. However, as this is the first attempt, we employ only limited types of contextual information in this paper. More potentially useful contextual information will be explored in the future. For Python projects, we collect the topics associated with each of them on GitHub. A topic on GitHub is a keyword that the Python project belongs to. Such keywords are generated by GitHub and chosen by developers, and thus can accurately describe the features of the corresponding projects. For TPLs, we collect keywords, authors, and TPL dependencies of each TPL from its installation wheel file. Specifically, we use the emails to identify different authors. The TPL dependencies are extracted from the setup.py file in the package and/or TPL description. 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 collected and then used by PyRec. More contextual information will be employed for experiments in the future.

Dataset Creation. To build a fair testbed for all competing approaches, following the same settings as [13], [17], we include only projects invoking 5 or more TPLs for evaluation. After removing projects with less than 5 TPLs, we have 12,421 projects in total in our dataset. Those projects invoke 963 distinct TPLs in total. There is a total number of 121,474 project-library interaction records, i.e., those TPLs are used 121,474 times in total by those projects. In terms of contextual information, the dataset has 73,277 pieces of records involving 9,675 extra entities corresponding to project topics, TPL authors, and TPL topics. The details are summarized in Table I.

[9]https://docs.github.com/en/rest/search#search-repositories
[10]https://pypi.org/

TABLE I
SUMMARY OF PROJECTS, TPLS, AND CONTEXTUAL INFORMATION

Data Type		Amount
Project-TPL Interaction	Python Project	12,421
	Third-party Library	963
	TPL Usage Information	121,474
Contextual Information	Project Topic Keyword	68,951
	TPL Author	431
	TPL Topic Keyword	1,420
	TPL Dependency	2,475

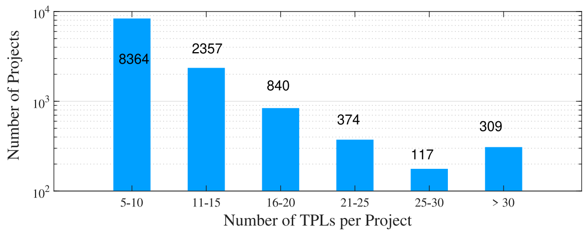

Fig. 5. TPL distribution.

The TPL distribution is depicted in Fig. 5. Each unique keyword, author, topic, etc., is treated as an individual entity (graph node) when creating the knowledge graph $\mathcal{KG}$. As introduced in Section III-A, 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.

Following the same experimental settings in [4], [13], [16], [17], we randomly remove rm TPLs from 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 $rm = 60\%$ means only 40% 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 ensure the comparability, the dataset with $rm = 60\%$ is a subset of the dataset with $rm = 40\%$, which is a subset of the dataset with $rm = 20\%$.

The removed TPLs constitute a test set and the remaining TPLs constitute a training set, the same as the settings in [4], [13], [16], [17]. 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 IV-I. To minimize the risk of label leakage, in the experiments, only those projects and TPLs in the training set are used to construct the KG. Besides, only contextual information that is relevant to those projects/TPLs in the training set is employed to construct the KG. As introduced in Section III, to generate the recommendation for a project, 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. It may be concerned that a new project or new TPL may not have too much contextual information in practice. However, PyRec does not require every project/TPL to have contextual information. In fact, it can build a simple KG solely based on the project-TPL interactions, which is equivalent to BG. Then, once contextual information is available, such information can be supplemented to KG and eventually fed to PyRec.

For each group of parameter settings, we conduct 50 experimental runs, i.e., executing the experiment 50 times with random removals, similar to [13] and [17]. Then, the performance averaged by arithmetic mean is reported. Considering the TPL scales invoked by those projects, this allows different TPLs to have roughly equal opportunity to be tested, similar to cross-validation used in [4]. Besides, a potential advantage is that the random removal allows different combinations of TPLs, which is overlooked by the cross-validation.

2) Implementation: We prototype PyRec in Python. For the other competing approaches [4], [13], [16], [17], we simply run their open-source codes with the Python dataset. In PyRec, 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 [36] to adaptively adjust the learning rate. For the other competing approaches, we keep the original parameters/settings reported in the corresponding papers and/or employed in the corresponding source codes. 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.

3) Metrics: Our objective is to propose an innovative approach to help Python developers effectively identify the most suitable TPLs. Therefore, we not only evaluate PyRec's ability to recommend libraries accurately but also measure PyRec's ability to recommend diverse TPLs. All metrics are widely used by researchers in not only the SE community but also the recommender system community. Specifically, the first four metrics are to measure the accuracy and the last one is to measure the diversity. Greater values for each metric indicate better performance.

- **Mean Precision (MP)** [4], [13], [38]: 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)** [4], [13], [14], [16]: 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)** [13], [17]: 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)** [4], [17]: 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)** [4], [13], [17]. 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.

B. Performance Comparison (RQ1)

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

- **LibRec** [16]: It is the first TPL recommendation approach. It combines CF and association rule mining to recommend useful TPLs for Java projects.
- **CrossRec** [4]: It is a CF-based approach designed for open-source Java projects.
- **LibSeek** [17]: It is an MF-based TPL recommendation approach facilitating Android app development.
- **GRec** [13]: 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 [4]. 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 [13], [17], so we set n to 5, 10, and 20, respectively. Please note that LibRec, CrossRec, LibSeek, and GRec can only use the project-TPL interactions to make recommendations. Thus, only such project-TPL interaction information in the training set is employed as their input. In contrast, PyRec can utilize not only project-TPL interaction information but also contextual information. Therefore, both of the above-mentioned information are employed as the input of PyRec. The potential threats to the conclusion validity will be discussed later in Section IV-I. Table II reports the performance of all competing approaches.

TABLE II
PERFORMANCE COMPARISON. DATA WITH UNDERLINES ARE THE BEST PERFORMANCE ACHIEVED BY EXISTING APPROACHES

Dataset	Approach	n=5					n=10					n=20				
		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
	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
	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
	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
	PyRec	**0.1106**	**0.3781**	**0.1711**	**0.3276**	**0.3857**	**0.0724**	**0.4844**	**0.1259**	**0.3424**	**0.4609**	**0.0455**	**0.5926**	**0.0845**	**0.3495**	**0.5553**
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
	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
	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
	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
	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**
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
	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
	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
	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
	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**

Please note that the *minimum advantages* (Min. Adv.) are calculated by comparing PyRec with the best performance achieved by state-of-the-arts (underlined).

The first observation is that PyRec achieves the best performance in every case. Taking $n = 10$ and $rm = 40\%$ as an example, PyRec achieves 0.1438, 0.4549, 0.2184, 0.5312, and 0.4994 in MP, MR, MF, MRR, and COV, respectively. Meanwhile, LibSeek is the second-best approach achieving 0.1169, 0.3598, 0.1646, and 0.4359 in MP, MR, MF, and MRR, respectively. GRec achieves the second-best performance in COV, with a value of 0.4968. Accordingly, PyRec outperforms LibSeek by 23.08%, 26.43%, 32.69%, and 21.88% in MP, MR, MF, and MRR, respectively. Although PyRec outperforms GRec by 0.52% in COV, it outperforms GRec by 78.92%, 65.14%, 84.48%, and 86.05% in MP, MR, MF, and MRR, respectively. The advantages of PyRec over LibRec and CrossRec are more significant. 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 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 of recommending 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 of being 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.

Cold start problem is an essential problem faced by recommendation systems. This paper focuses on general TPL recommendation scenarios, the cold start problem will be studied in the future. However, the results reported in Table II demonstrate that PyRec has better ability than the other approaches to address the cold start problem. For example, when $rm = 60\%$, each project has only 40% used TPLs, indicating that there is only limited historical data used for recommendation. It is easy to see from Table II that PyRec has greater advantages in every metric than the other approaches. For example, when $rm = 20\%$ and $n = 20$, PyRec's advantages in MR are 51.69%, 77.81%, 11.45%, and 9.95%, respectively, against LibRec, CrossRec, LibSeek, GRec, and PyRec. When $rm = 60\%$, the corresponding advantages become 283.59%, 64.08%, 49.51%, and 68.79%, respectively. That is because when rm increases, less information can be employed to make the recommendation. This limits their performance. Therefore, the MRs of all approaches decrease accordingly. However, PyRec's MR decreases slightly from 0.5926 to 0.5324 while others decrease significantly, thanks to PyRec's ability to incorporate contextual information to supplement the recommendation. This observation evidences that PyRec is more capable of handling the cold start problem. It also

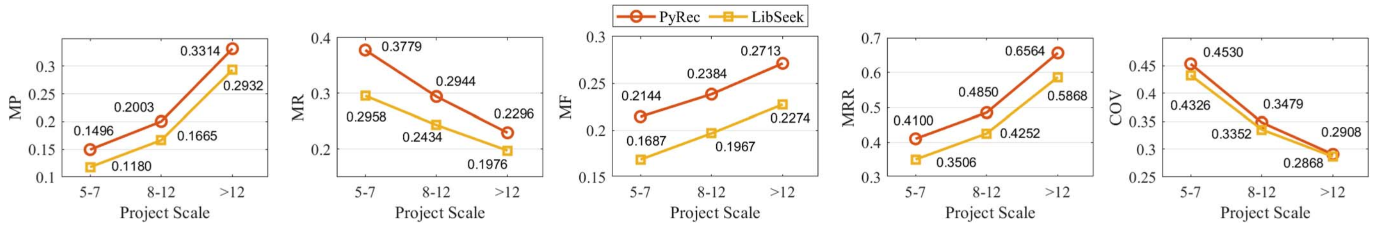

Fig. 6. Impact of project scales ($rm = 40\%, n = 5$).

confirms our statement made in Section I 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 an early development stage (e.g., $rm = 60\%$).

C. 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 II**, LibSeek has the best performance across all existing approaches. Thus, we employ LibSeek only for comparison. The results are shown in **Fig. 6**.

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 twofold. 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 [17]. 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%, respectively, 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 II, i.e., PyRec is more useful for projects at the early development stage.

D. 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 [13]. 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 information, denoted as PyRec$_{c1}$, separately. **Fig. 7** 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 I 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 II 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.

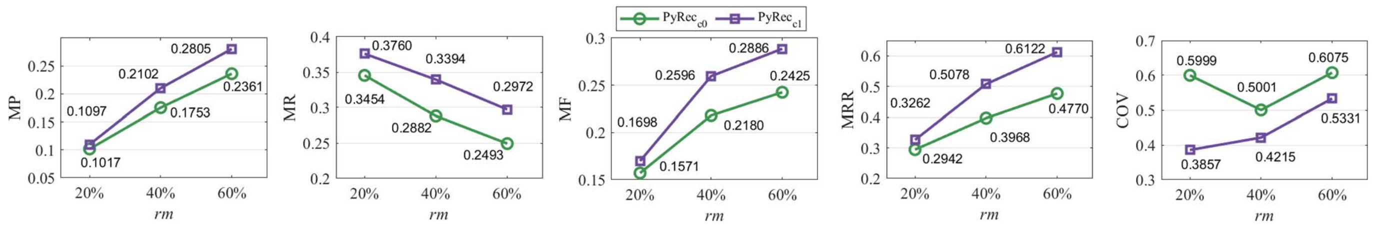

Fig. 7. Impact of contextual information ($n = 5$).

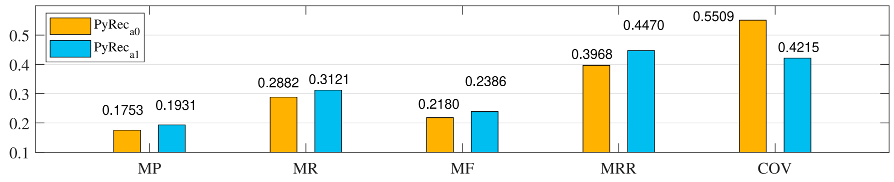

Fig. 8. Impact of attention mechanism ($rm = 40\%, n = 5$).

E. Usefulness of Attention Mechanism (RQ4)

Now we investigate whether the adoption of attention mechanisms in Section III (Eqs. (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. 8** 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 III. Similar to the phenomena observed in Fig. 7, PyRec$_{a0}$ performs better in COV than PyRec. The underlay reason is also the same.

F. 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. 9** reports the average performance achieved by each approach when $rm = 40\%$ and $n = 5$. 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%.

We can find that along with the increase of k, PyRec's recommendation accuracy becomes better in all cases.

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 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. 10**. 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 III-C, 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.

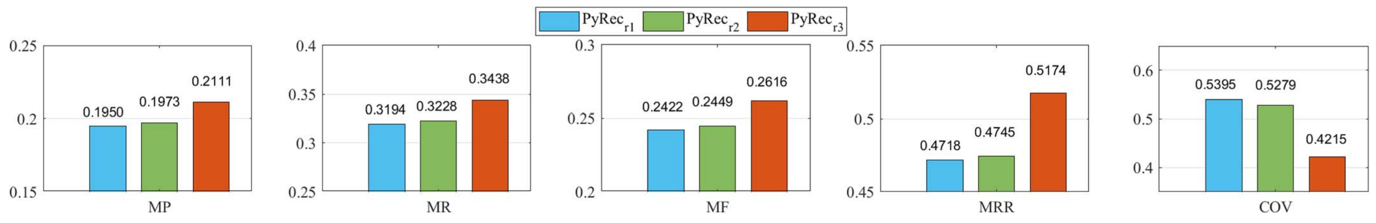

Fig. 9. Impact of relation embedding size (k) ($rm = 40\%, n = 5$).

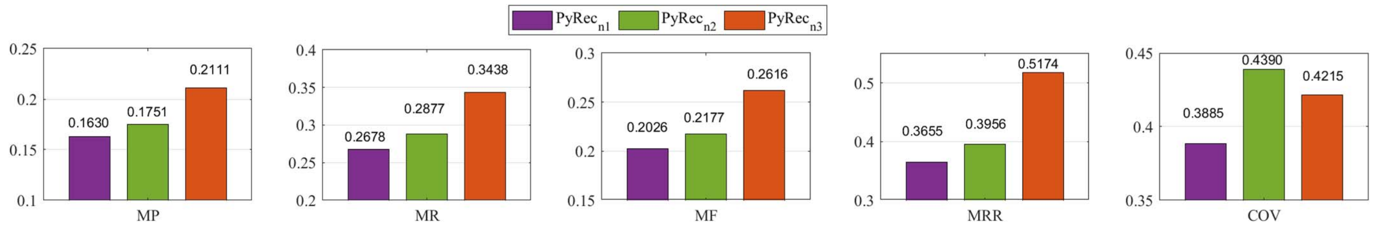

Fig. 10. Impact of node embedding size (d) ($rm = 40\%, n = 5$).

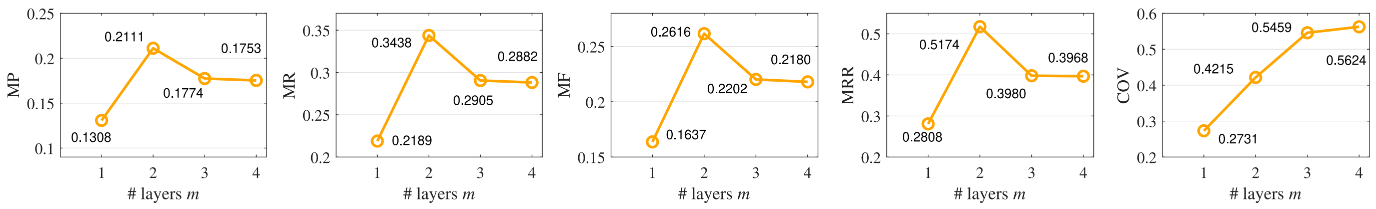

Fig. 11. Impact of number of GNN layers (m) ($rm = 40\%, n = 5$).

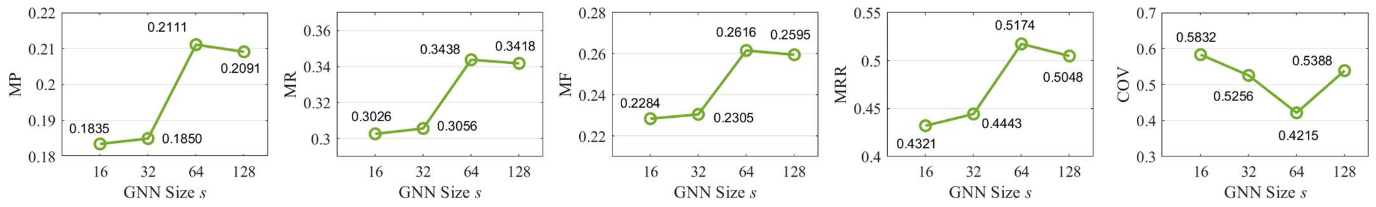

Fig. 12. Impact of GNN layer size (s) ($rm = 40\%, n = 5$).

Based on the above observations we can find that a greater value of d results in better performance in MP, MR, MF, and MRR.

By comparing Figs. 9 and 10, 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 III-D, 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. 11**.

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 [13]. 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 of being 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. 12, 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

TABLE III
PERFORMANCE COMPARISON WITH DIFFERENT DATASET SCALES

Dataset Scale	n=5					n=10					n=20					Training Time
	MP	MR	MF	MRR	COV	MP	MR	MF	MRR	COV	MP	MR	MF	MRR	COV	
Scale=40%	0.1873	0.3104	0.2336	0.4257	0.4409	0.1369	0.4439	0.2092	0.4448	0.4574	0.0910	0.5725	0.1570	0.4516	0.5363	31.3 mins
Scale=70%	0.1954	0.3236	0.2436	0.4535	0.4210	0.1384	0.4473	0.2114	0.4704	0.4397	0.0911	0.5719	0.1571	0.4768	0.5162	64.1 mins
Scale=100%	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	115.2 mins

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. 11. 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.

G. Impact of Knowlege Graph Scale

Our PyRec makes TPL recommendations based on the project-TPL knowledge graph. The more projects and/or TPLs involved, the larger the corresponding knowledge graph is. Now, we study the impact of graph scales on PyRec's performance. Specifically, we build another two knowledge graphs by randomly choosing 40% and 70% projects from the original dataset. Then, we build the graphs with chosen projects and corresponding TPLs and contextual information. The rm is set to 40%, and n is set to 5, 10, and 20, respectively. The results are reported in Table III.

We can find that when more projects are included in the knowledge graph, PyRec has a better performance in MP, MR, MF, and MRR. This is expected as more projects involve more TPLs and more contextual information. Then, PyRec can make better use of that information to make more accurate recommendations. However, the performance in COV is slightly turbulent when the dataset scale increases.

The training time is 31.3 mins, 64.1 mins, and 115.2 mins, respectively, with different dataset scales. This is acceptable as PyRec can be trained once and used several times. Given a project, PyRec takes 1.2 s, 2.3 s, and 2.6 s, respectively, to make recommendations when the dataset scale is 40%, 70%, and 100%, respectively.

> PyRec is capable of making near real-time responses, making it practical in real-world applications.

H. Case Study

1) Usefulness of Contextual Information: One of the innovations of PyRec is the utilization of contextual information. We employ project #1048 OpenMatch[11] as an example to demonstrate the usefulness of contextual information. We randomly remove 40% of TPLs used by OpenMatch and use those left (shown in the white disks in Fig. 13) as input for PyRec. We ask PyRec to recommend 5 TPLs with/without utilizing contextual information. We can find that when using contextual information, PyRec recommends three correct TPLs with IDs of 4, 11, and 8. In contrast, without using the contextual information, PyRec recommends only two correct TPLs whose IDs are 8 and 4, respectively.

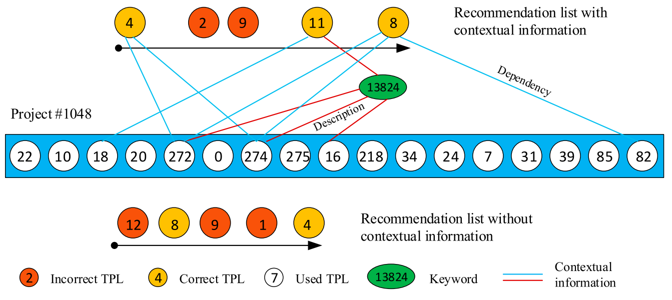

Fig. 13. Exemplary recommendation for project #1048 in our dataset.

Taking TPL #11 (DeepPavlov[12]) as an example, without contextual information, PyRec overlooks this TPL. When using contextual information, PyRec puts DeepPavlov in the fourth position in the recommendation list, because 1) DeepPavlov has a dependency relationship with TPL #18 (NLTK[13]) and 2) DeepPavlov is described by keyword #13824 (NLP) which also describes TPLs #272 (Pytorch Lightning[14]), #274 (PyThaiNLP[15]), and #16 (Adaptor Transformer[16]). The above extra relationships between TPL DeepPavlov and used TPLs by OpenMatch indicate that DeepPavlov is potentially useful for OpenMatch, and thus is included in the list.

> Utilizing contextual information can improve PyRec's recommendation performance.

2) Recommendation for New Projects: Now we study PyRec's performance when recommending TPLs for new projects. Specifically, we randomly choose 80% of projects and relevant TPLs and contextual information as the training set to create the knowledge graph used by PyRec. We take the other 20% projects as the test set. For each project in the test set, we randomly choose 60% used TPLs as input and check if PyRec can successfully recommend the other 40% TPLs.

[11]https://github.com/thunlp/OpenMatch

[12]https://deeppavlov.ai/
[13]https://www.nltk.org/
[14]https://pypi.org/project/pytorch-lightning/
[15]https://pythainlp.github.io/
[16]https://pypi.org/project/adapter-transformers/

TABLE IV
PERFORMANCE ON NEW PROJECTS

List length	MP	MR	MF	MRR	COV
n=5	0.2311	0.3763	0.2864	0.5616	0.5147
n=10	0.1546	0.4865	0.2347	0.5736	0.6260
n=20	0.0980	0.5979	0.1684	0.5781	0.7421

Table IV reports the averaged performance. Compared the results reported in Table IV with the results reported earlier in Table II we can find that PyRec still has a good performance when working with new projects, indicating its good adaptability. Please note that we randomly remove TPLs from each new project to construct the test set. Unfortunately, the limitation is that the random removal may not adequately evaluate PyRec's performance in practice, and thus can be further improved in the future.

3) Rare TPL Recommendation: The same as the findings reported in [17], the TPL usage is significantly biased. For example, the top 1% most popular TPLs in our dataset dominate 69.31% TPL usage across those projects. A great portion of rare TPLs do not have too much usage. To study PyRec's ability to recommend rare TPLs, we rank all TPLs according to their popularities in the dataset and take 200 TPLs ranked from 501 to 700 as examples. For each project in the dataset, we randomly remove its 40% TPLs and run PyRec to recommend a list with 10 TPLs. Then, we measure how many times those rare TPLs are included in the test set and the recommendation lists. We repeat the experiments 10 times and report the grand total for each TPL in Fig. 14.

It can be found that PyRec can successfully recommend rare TPLs. However, most of the time, PyRec recommends less rare TPLs than expected, indicated by the fact that a TPL appears more times in the test set than the recommendation lists. This is acceptable as recommending rare items is still challenging for existing recommender systems [26]. The second observation is that PyRec tends to recommend slightly less rare TPLs when utilizing contextual information. This is because popular TPLs usually have more contextual information and thus are much easier to be recommended by PyRec. However, if a rare TPL has contextual information, it has more opportunity to be recommended by PyRec. Taking TPL JikanPy[17] (#459, ranked 521) as an example, it has two topic keywords *cryptography* (#14328) and *security* (#13859) in PyPI, which connects it to the other 7 TPLs. Besides, it has a dependency relationship with a popular TPL *Cryptography* (#136, ranked 52). The above contextual information provides many connections in the knowledge graph. As a result, it is recommended 158 times when contextual information is included, compared with 95 times in the test set and 132 times when making recommendations without contextual information.

I. Threats to Validity

Internal Threats. The first threat comes from the dataset scale. Given the huge number of available projects and TPLs,

[17]https://pypi.org/project/jikanpy/

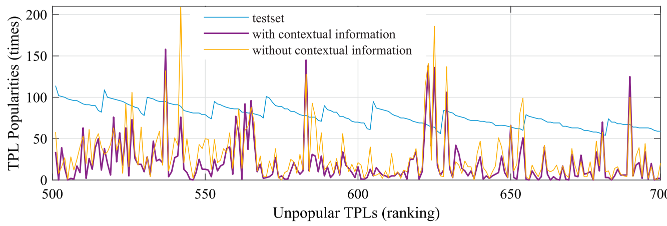

Fig. 14. Rare TPL recommendation.

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. In addition, we experimentally evaluated PyRec's performance with different project scales, i.e., 40%, 70%, and 100% of collected projects. Therefore, the bias may exist but is not significant. The second threat comes from the implementation of PyRec and other competing approaches. To minimize this threat, we made PyRec publicly available for 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 [20] - 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 [16], open-source projects [4], and Android mobile app development [13], [17], respectively. The results reported in Table II 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 [13], [17] 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

[4] and LibSeek [17] 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 [4], [13], [16], [17] 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. Thus, the lack of project evolution information poses 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 IV-D and Section IV-E, 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 [4], [13], [16], [17], 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.

V. Related Work

Recommendation techniques have been widely adopted to facilitate software development, maintenance, and evolution, such as defect identification [39], developer recommendation [40], [41], [42], [43], API/code snippet recommendation [27], [44], third-party library recommendation [13], [17], permission recommendation [45], etc. Among them, our work is closely related to API recommendations and TPL recommendations.

A. API Recommendation

As suggested by its name, the API recommendation is to provide developers with useful APIs and/or code snippets, aiming at improving developers' coding efficiency [46], [47], [48], [49].

Many researchers have tried to recommend APIs based on contextual information. To name a few, Thung et al. recommended APIs to implement software features based on APIs' textual descriptions about those features [46]. Similarly, Huang et al. focused on mapping developers' demand descriptions to APIs' structured feature descriptions to find more suitable APIs [47]. He et al. considered more contextual information when recommending Python APIs. They proposed a random forest-based approach utilizing data flow, token similarity, and token co-occurrence [21].

The API usage information is also useful for finding useful APIs. Liu et al. constructed function call graphs and then recommended APIs based on the API usage paths distilled from those graphs [49]. Similarly, Xie et al. recommended new APIs by distilling hierarchical contextual information from the project's call graph [48]. However, the above two methods suffer from efficiency issues as the involved graphs are usually too complex. Different from them, Nguyen et al. employed a CF-based recommendation technique to find useful APIs for open-source projects [38]. Specifically, they collected a large number of projects and extracted the historical API usage information, based on which new APIs were recommended for those projects. Zhao et al. utilized both textual descriptions and TPL usage information to make API recommendations. They proposed APIMatchmaker to recommend APIs for Android app development [27]. Wu et al. proposed a neural framework leveraging multi-model fusion and multi-task learning techniques to recommend Web APIs [50]. Gong et al. [51] 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, i.e., it works at a fine-grained level from the software development perspective.

B. TPL Recommendation

Different from API recommendation, the TPL recommendation is to find entire TPLs that are potentially useful for software development, which works at a coarse-grained level. Although many efforts have been devoted to identifying TPL usage patterns [9], [14], [52], there is still a lack of effective solutions to recommend TPLs.

To tackle this issue, Thung et al. combined association rule mining and TPL-based CF to recommend TPLs for Java projects [16]. As this is the first attempt at the TPL recommendation issue, the performance is limited. Later, Nguyen et al. combined project-based CF and TPL-based CF to recommend TPLs for open-source Java projects [4]. This further improved the recommendation performance, especially the precision of the recommended TPLs. However, He et al. found that such methods tend to recommend popular TPLs, which is indeed not useful enough for mobile app development [17]. To solve this issue, they proposed an MF-based approach that can diversify the recommended TPLs. Very recently, Li et al. proposed GRec which employs GNN to recommend TPLs for Android app development based on the app-library graph [13]. GRec is capable of distilling transitive information from BG consisting of TPLs and Android apps and thus has better performance. However, all of the above approaches utilize only the TPL usage information to find useful TPLs. None of them can utilize the contextual information that has been widely used in the API recommendation area.

Different from all the above approaches, PyRec proposed in this paper 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.

C. Python Library

As an emerging programming language, Python has been widely used in recent years. Many efforts have been devoted to the study of Python libraries. Those studies can be categorized into two categories, including the TPL/API evolution [1], [2], [19] and the TPL dependency analysis [53], [54], [55], [56].

To name a few studies in TPL/API evolution, 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 [57]. Zhang et al. investigated the API evolution in Python libraries and detected compatibility issues caused by such API evolution [2]. 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 [1]. Rubei et al. [19] integrated end-user feedback to provide developers with TPL evolution suggestions, i.e., whether to keep or discard a used TPL.

In terms of the TPL dependency analysis, Wang et al. studied the TPL dependency issues in Jupyter Notebooks [54]. 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 [55]. Ying et al. proposed Watchman to automatically detect dependency conflicts among Python TPLs for the PyPI ecosystem [56].

Different from the above-mentioned studies, this paper focuses on recommending potentially useful TPLs to facilitate Python project development. However, as PyRec can leverage various kinds of contextual information, the TPL dependencies and evolution information identified by the above studies could be a fruitful supplement for PyRec.

D. Knowledge Graph in Software Engineering

The knowledge graph can effectively model complex relationships between different entities and thus was widely adopted to solve software engineering problems, such as runtime environment configuration, API analysis, bug fixing, and weakness analysis [58].

To name a few, Cheng et al. found that restoring the runtime environment for open-source Python projects is usually challenging [37]. To solve this issue, they designed a KG based on massive Python projects and Python third-party packages. Then, they proposed a heuristic graph traversal algorithm to infer the compatible runtime environment for the target Python project. To help developers compare APIs' commonalities and differences, Liu et al. proposed APIComp based on KG to automatically extract API knowledge from API reference documentation [59]. APIComp can compare API classes and/or methods from different perspectives. The misuse of APIs is a critical problem for software development. To solve this problem, Ren et al. detected API misuses against the API caveat knowledge [60]. Such knowledge is captured from an API-constraint knowledge graph generated with API reference documents. Cheng proposed a KG-based approach to utilize the deep semantic and structural relationships possessed in multi-source software projects [61]. Based on the generated KG, developers can effectively search for useful bug-fix knowledge in the Q&A manner. To enable the common weakness enumeration (CWE) for software maintenance, Han et al. mapped software weaknesses and their relations as a knowledge graph [62]. Then, they developed a knowledge representation learning method to embed the graph into a semantic vector space, based on which three reasoning tasks, including CWE link prediction, CWE triple classification, and common consequence prediction were enabled. Zhao et al. proposed HDSKG, an automatic method, to discover domain-specific concepts and their relation triples from web pages [63]. They constructed a domain-specific knowledge graph by analyzing web content on Stack Overflow.

Different from the above-mentioned studies, this paper employs KG to enable the TPL recommendation for software development, which provides a new adoption of KG in the software engineering field.

VI. Discussion and Future Work

There are a few problems that can be further studied and addressed in the future.

The first problem is to evaluate PyRec's applicability, i.e., investigating PyRec's adaptability to projects developed in other languages. This can be achieved by collecting projects and TPLs developed in different languages, like Java, .NET, etc. As different programming languages have different characteristics, it is better to construct an individual dataset for each language.

The second problem is to extend PyRec to recommend TPL updates for software projects. Specifically, given a software application, PyRec identifies if any TPLs can be updated and if yes, recommends developers the specific versions. It is challenging as the TPL update usually leads to compatibility issues [64]. Another challenge is the current dataset used by PyRec does not have TPL version information. One potential solution is to extend the dataset to include all TPL versions and the compatibility information for each version. Meanwhile, the knowledge graph should be extended to incorporate such new information.

The third problem is to further improve PyRec's performance. One potential solution is to incorporate more projects, TPLs, and contextual information to enrich the knowledge graph. The second potential solution is to update the graph embedding technique used by PyRec. Although TransR has a good performance, new embedding techniques like TransH, TransE, TransD, ComplEx, etc., can be explored to further improve PyRec's performance.

The fourth problem is the cold start problem, which is a common problem faced by modern recommender systems [65]. This is challenging as limited information is available for PyRec to construct the knowledge graph when a project just starts. To tackle this issue, new methodologies like dropout net [66] and aligning distillation [67] can be explored.

The last problem is to improve the evaluation methodology to better investigate PyRec's performance. In Section IV, we followed the settings in [4], [13], [16], [17] to randomly remove a few TPLs to establish the test set. Unfortunately, there is still a threat. For example, if too many popular TPLs are removed, the recommendation complexity could be reduced. However, creating a suitable test set is challenging. To tackle this problem, one potential solution is to evaluate PyRec in the wild. Specifically, collecting a few new projects and making recommendations for each project by PyRec. Then, their developers can be contacted to check if the recommended TPLs are indeed useful.

VII. Conclusion

In this paper, we proposed an 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.

References

[1] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2020, pp. 233–244.

[2] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do Python framework APIs evolve? An exploratory study," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 81–92.

[3] 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 Softw. Eng.*, vol. 25, no. 1, pp. 755–789, 2020.

[4] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "CrossRec: Supporting software developers by recommending third-party libraries," *J. Syst. Softw.*, vol. 161, 2020, Art. no. 110460.

[5] Y. Zhang et al., "Detecting third-party libraries in Android applications with high precision and recall," in *Proc. 25th IEEE Int. Conf. Softw. Anal., Evolution Reeng.*, 2018, pp. 141–152.

[6] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA: ACM, 2016, pp. 356–367.

[7] M. Li et al., "LibD: Scalable and precise third-party library detection in Android markets," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, 2017, pp. 335–346.

[8] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and accurate detection of third-party libraries in Android apps," in *Proc. 38th Int. Conf. Softw. Eng. Companion (ICSE)*, New York, NY, USA: ACM, 2016, pp. 653–656.

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

[10] M. Lamothe and W. Shang, "When APIs are intentionally bypassed: An exploratory study of API workarounds," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, vol. 2020, 2020, pp. 912–924.

[11] X. Zhan et al., "Automated third-party library detection for Android applications: Are we there yet?" in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2020, pp. 919–930.

[12] T. Ki, C. M. Park, K. Dantu, S. Y. Ko, and L. Ziarek, "Mimic: UI compatibility testing system for Android apps," in *Proc. 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 246–256.

[13] B. Li et al., "Embedding app-library graph for neural third party library recommendation," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2021, pp. 466–477.

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

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

[16] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *Proc. 20th Work. Conf. Reverse Eng. (WCRE)*, 2013, pp. 182–191.

[17] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang, "Diversified third-party library prediction for mobile app development," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 150–165, Jan. 2022.

[18] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin, "Incorporating contextual information in recommender systems using a multidimensional approach," *ACM Trans. Inf. Syst.*, vol. 23, no. 1, pp. 103–145, 2005.

[19] 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 *Proc. IEEE Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 817–821.

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

[21] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, and B. Xu, "PyART: Python API recommendation in real-time," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 1634–1645.

[22] X. Wang, X. He, Y. Cao, M. Liu, and T.-S. Chua, "KGAT: Knowledge graph attention network for recommendation," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2019, pp. 950–958.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Proc. 14th Eur. Conf. Comput. Vis. (ECCV),* Amsterdam, the Netherlands, Proc., Part IV 14, New York, NY, USA: Springer-Verlag, 2016, pp. 630–645.

[24] W. Yu, Z. Zhang, and Z. Qin, "Low-pass graph convolutional network for recommendation," in *Proc. AAAI Conf. Artif. Intell.*, 2022, vol. 36, no. 8, pp. 8954–8961.

[25] Y. Pang et al., "Heterogeneous global graph neural networks for personalized session-based recommendation," in *Proc. 15th ACM Int. Conf. Web Search Data Mining*, 2022, pp. 775–783.

[26] C. Gao et al., "A survey of graph neural networks for recommender systems: Challenges, methods, and directions," *ACM Trans. Recommender Syst.*, vol. 1, no. 1, pp. 1–51, 2023.

[27] Y. Zhao, L. Li, H. Wang, Q. He, and J. Grundy, "APIMatchmaker: Matching the right APIs for supporting the development of Android apps," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 113–130, Jan. 2023.

[28] S. Ji, S. Pan, E. Cambria, P. Marttinen, and S. Y. Philip, "A survey on knowledge graphs: Representation, acquisition, and applications," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 2, pp. 494–514, Feb. 2022.

[29] Z. Li, H. Liu, Z. Zhang, T. Liu, and N. N. Xiong, "Learning knowledge graph embedding with heterogeneous relation attention networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 8, pp. 3961–3973, Aug. 2022.

[30] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 2181–2187.

[31] Q. Wang, Z. Mao, B. Wang, and L. Guo, "Knowledge graph embedding: A survey of approaches and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 12, pp. 2724–2743, Dec. 2017.

[32] J. Zhuo et al., "Tiger: Transferable interest graph embedding for domain-level zero-shot recommendation," in *Proc. 31st ACM Int. Conf. Inf. Knowl. Manage.*, 2022, pp. 2806–2816.

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

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

[35] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, "Neural graph collaborative filtering," in *Proc. 42nd Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2019, pp. 165–174.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Poster Int. Conf. Learn. Representations (ICLR)*, 2015, pp. 1–15.

[37] W. Cheng, X. Zhu, and W. Hu, "Conflict-aware inference of Python compatible runtime environments with domain knowledge graph," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2022, pp. 451–461.

[38] 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 *Proc. 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 1050–1060.

[39] Y. Zhao et al., "ReCDroid: Automatically reproducing Android application crashes from bug reports," in *Proc. 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 128–139.

[40] L. Ye, H. Sun, X. Wang, and J. Wang, "Personalized teammate recommendation for crowdsourced software developers," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 808–813.

[41] L. Bao, X. Xia, D. Lo, and G. C. Murphy, "A large scale study of long-time contributor prediction for GitHub projects," *IEEE Trans. Softw. Eng.*, vol. 47, no. 6, pp. 1277–1298, Jun. 2021.

[42] 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 *Proc. IEEE Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 630–640.

[43] X. Xie, X. Yang, B. Wang, and Q. He, "DevRec: Multi-relationship embedded software developer recommendation," *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4357–4379, Nov. 2022.

[44] Y. Zhao, L. Li, X. Sun, P. Liu, and J. Grundy, "Icon2Code: Recommending code implementations for Android GUI components," *Inf. Softw. Technol. (IST)*, vol. 138, pp. 106619–106630, Oct. 2021.

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

[46] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 290–300.

[47] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-API knowledge gap," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 293–304.

[48] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "HiRec: API recommendation using hierarchical context," in *Proc. 30th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 369–379.

[49] X. Liu, L. Huang, and V. Ng, "Effective API recommendation without historical software repositories," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 282–292.

[50] H. Wu, Y. Duan, K. Yue, and L. Zhang, "Mashup-oriented web API recommendation via multi-model fusion and multi-task learning," *IEEE Trans. Services Comput.*, vol. 15, no. 6, pp. 3330–3343, Nov./Dec. 2022.

[51] W. Gong et al., "DAWAR: Diversity-aware Web APIs recommendation for mashup creation based on correlation graph," in *Proc. 45th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2022, pp. 395–404.

[52] M. A. Saied and H. Sahraoui, "A cooperative approach for combining client-based and library-based API usage pattern mining," in *Proc. IEEE 24th Int. Conf. Program Comprehension (ICPC)*, Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–10.

[53] E. Horton and C. Parnin, "DockerizeMe: Automatic inference of environment dependencies for Python code snippets," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 328–338.

[54] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of Jupyter notebooks," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 1622–1633.

[55] H. Ye, W. Chen, W. Dou, G. Wu, and J. Wei, "Knowledge-based environment dependency inference for Python programs," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 1245–1256.

[56] Y. Wang et al., "Watchman: Monitoring dependency conflicts for Python library ecosystem," in *ACM/IEEE 42nd Int. Conf. Softw. Eng. (ICSE)*, 2020, pp. 125–135.

[57] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *Proc. Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 72–83.

[58] G. Tamasauskaitė and P. Groth, "Defining a knowledge graph development process through a systematic review," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 1–40, 2023.

[59] Y. Liu, M. Liu, X. Peng, C. Treude, Z. Xing, and X. Zhang, "Generating concept based API element comparison using a knowledge graph," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 834–845.

[60] X. Ren et al., "API-misuse detection driven by fine-grained API-constraint knowledge graph," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 461–472.

[61] C. Zhou, "Intelligent bug fixing with software bug knowledge graph," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA: ACM, 2018, pp. 944–947.

[62] Z. Han, X. Li, H. Liu, Z. Xing, and Z. Feng, "DeepWeak: Reasoning common software weaknesses via knowledge graph embedding," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, 2018, pp. 456–466.

[63] X. Zhao, Z. Xing, M. A. Kabir, N. Sawada, J. Li, and S.-W. Lin, "HDSKG: Harvesting domain specific knowledge graph from content of webpages," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, 2017, pp. 56–67.

[64] Y. Wang et al., "An empirical study of usages, updates and risks of third-party libraries in Java projects," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution (ICSME)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 35–45.

[65] B. Lika, K. Kolomvatsos, and S. Hadjiefthymiades, "Facing the cold start problem in recommender systems," *Expert Syst. Appl.*, vol. 41, no. 4, pp. 2065–2073, 2014.

[66] M. Volkovs, G. Yu, and T. Poutanen, "DropoutNet: Addressing cold start in recommender systems," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 4957–4966.

[67] F. Huang, Z. Wang, X. Huang, Y. Qian, Z. Li, and H. Chen, "Aligning distillation for cold-start item recommendation," in *Proc. 46th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2023, pp. 1147–1157.