Knowledge-Based Environment Dependency Inference for Python Programs

Hongjie Ye
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
yehongjie19@otcaix.iscas.ac.cn

Wei Chen*†
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
wchen@otcaix.iscas.ac.cn

Wensheng Dou†
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
wsdou@otcaix.iscas.ac.cn

Guoquan Wu†
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
gqwu@otcaix.iscas.ac.cn

Jun Wei
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
wj@otcaix.iscas.ac.cn

ABSTRACT

Besides third-party packages, the Python interpreter and system libraries are also critical dependencies of a Python program. In our empirical study, 34% programs are only compatible with specific Python interpreter versions, and 24% programs require specific system libraries. However, existing techniques mainly focus on inferring third-party package dependencies. Therefore, they can lack other necessary dependencies and violate version constraints, thus resulting in program build failures and runtime errors.

This paper proposes a knowledge-based technique named PyEGo, which can automatically infer dependencies of third-party packages, the Python interpreter, and system libraries at compatible versions for Python programs. We first construct the dependency knowledge graph PyKG, which can portray the relations and constraints among third-party packages, the Python interpreter, and system libraries. Then, by querying PyKG with extracted program features, PyEGo constructs a program-related sub-graph with dependency candidates of the three types. It finally outputs the latest compatible dependency versions by solving constraints in the sub-graph. We evaluate PyEGo on 2,891 single-file Python gists, 100 open-source Python projects and 4,836 jupyter notebooks. The experimental results show that PyEGo achieves better accuracy, 0.2x to 3.5x higher than the state-of-the-art approaches.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**; **Maintaining software**.

KEYWORDS

Python, environment dependency inference, version constraint, knowledge graph

ACM Reference Format:
Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-Based Environment Dependency Inference for Python Programs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510127

1 INTRODUCTION

Python programs depend on third-party packages (i.e., Python libraries), the Python interpreter, and system libraries. Missing and incompatible environment dependencies can result in program build failures and runtime errors.

Developers need to infer environment dependencies for Python programs due to the following reasons. First, many open-source Python programs (e.g., Python gists and Jupyter Notebooks in GitHub) do not explicitly declare their dependencies or miss some dependencies [23, 38]. Second, Python libraries are usually frequently updated, may become deprecated, or are removed due to security issues [30]. Third, a program migrating from one environment to another may fail due to overlooking required system libraries, which are usually not explicitly documented.

Besides third-party packages, Python programs also depend on specific system libraries and the Python interpreter. In our empirical study on 100 gists sampled from HG2.9k [24], we find that 34 gists

*Wei Chen is the corresponding author.
†Wei Chen, Wensheng Dou and Guoquan Wu are also affiliated with Nanjing Institute of Software Technology and University of Chinese Academy of Sciences, Nanjing, China.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510127

suffer from runtime errors due to incompatible Python interpreter versions, and 24 gists fail due to missing system libraries. Despite the importance of system library and Python interpreter dependencies, we find that only half of our investigated programs declare compatible Python versions, and only a quarter of projects document dependent system libraries. Therefore, an effective approach is still required to "*dig and build the full required dependencies* [14]."

Some studies have tried to address Python program environment dependency issues. The prior work [23, 24] infers third-party packages and system libraries for single-file Python programs, but they only recommend the latest package versions without concerning version constraints. *SnifferDog* [37] infers dependent Python libraries for Jupyter notebooks based on API usage analysis, but it neglects system libraries and the Python interpreter. Pipreqs [6], a popular open-source tool, only focuses on third-party packages and does not pay attention to system libraries and the Python interpreter.

This paper proposes a knowledge-based technique named PyEGo, which can automatically infer dependencies at compatible versions for Python programs. PyEGo considers dependencies of third-party packages, the Python interpreter, and system libraries. Based on a thorough analysis of the knowledge required for environment dependency inferences, we first construct the dependency knowledge graph PyKG, which can portray the three types of dependencies and their relations. Then, facilitated with PyKG, PyEGo infers environment dependencies for a Python program via static program analysis and constraint solving. It extracts program features and takes them as inputs to query PyKG for the candidate versions of possible dependencies. The dependency candidates form a program-related dependency sub-graph with their interrelations. PyEGo further infers the latest compatible dependency versions within the sub-graph by solving constraints among the candidates. In essence, PyEGo is an exploratory step towards automating dependency inferences for Python program.

We evaluate PyEGo on HG2.9K [24], containing 2,891 single-file Python programs, by resolving `ImportErrors` with inferred environment dependencies. In addition, we evaluate PyEGo on 100 more complex Python projects and 4,836 jupyter notebooks. Overall, PyEGo achieves 46.14%, 62% and 60.90% accuracy, respectively, which is 0.2x to 3.5x higher than the state-of-the-art approaches.

In summary, this work makes the following contributions.

- We propose a dependency knowledge graph and its construction approach. PyKG can portray the relations among third-party packages, the Python interpreter, and system libraries.
- We propose a knowledge-based environment dependency inference technique PyEGo, which regards constraints among a program and its dependencies and can infer the latest compatible dependency versions of three dependency types, i.e., third-party packages, the Python interpreter, and system libraries.
- The evaluations on HG2.9K, 100 open-source projects and 4,836 jupyter notebooks reveal that PyEGo is more effective than state-of-the-art approaches.

2 MOTIVATION

In this section, we perform an empirical study for investigating the prevalence of environment dependency issues. Then, we use an example to analyze the challenges of environment dependency inference for Python programs. Finally, we present an overview of our approach addressing the challenges.

2.1 Empirical Study

Our small-scale empirical study concentrates on dependencies issues relating to the Python interpreter and system libraries because the prior work [24, 37] has confirmed that Python packages are indispensable to program builds and executions. This empirical study is dedicated to answering the following two research questions.

- **RQ1**: To what extent do open-source Python projects provide documented environment dependencies? What kinds of dependencies are documented?
- **RQ2**: Besides Python packages, do the Python interpreter and system libraries affect Python program builds and executions seriously?

To answer RQ1, we randomly sample 100 popular Python projects on Github with more than 1000 stars. We identify their dependency declaration files and investigate what kinds of dependencies they document. We find that (1) 79 out of 100 projects declare third-party package dependencies, (2) 51 projects declare compatible Python interpreter versions, and (3) only 27 projects declare system library dependencies. In addition, We categorize dependency declaration files. `Requirements.txt` is the most popular declaration file (54 of 100 projects), but they only record third-party package dependencies. `Dockerfile`, `Pipfiles`, and conda YAML files can record dependencies of the Python interpreter and system libraries, but only 26 projects use declaration files of these types.

Findings: Most open-source Python projects document their third-party packages dependencies. However, only half of the investigated projects declare their compatible Python versions, and even worse, only a quarter of projects declare system library dependencies. Moreover, although `requirements.txt` is most prevalent, they do not document the dependencies of the Python interpreter and system libraries.

To answer RQ2, we randomly sample 100 gists from HG2.9K [24]. We manually construct runtime environments for these gists and investigate the encountered problems. Only 42 out of 100 gists can execute successfully by simply installing the third-party dependent packages. For the remaining 58 gists, (1) 34 gists encounter runtime errors due to incompatible Python versions because 19 and 15 gists are only compatible with Python2 and Python3, respectively. (2) 24 gists fail to build or execute due to missing system libraries. In addition, we investigate the top 30 voted Python questions in Stack Overflow. We find that (1) 11 of 30 questions are relevant to incompatible Python interpreter versions, and (2) 6 of 30 questions are relevant to third-party package installation failures caused by missing some system libraries.

Findings: The Python interpreter and system libraries are critical dependencies of Python programs. Unfortunately, more than half of the gists experience build failures and execution errors due to incompatible Python versions or missing system libraries.

In summary, besides third-party packages, the Python interpreter and system libraries are also indispensable, but few techniques concern them in practice.

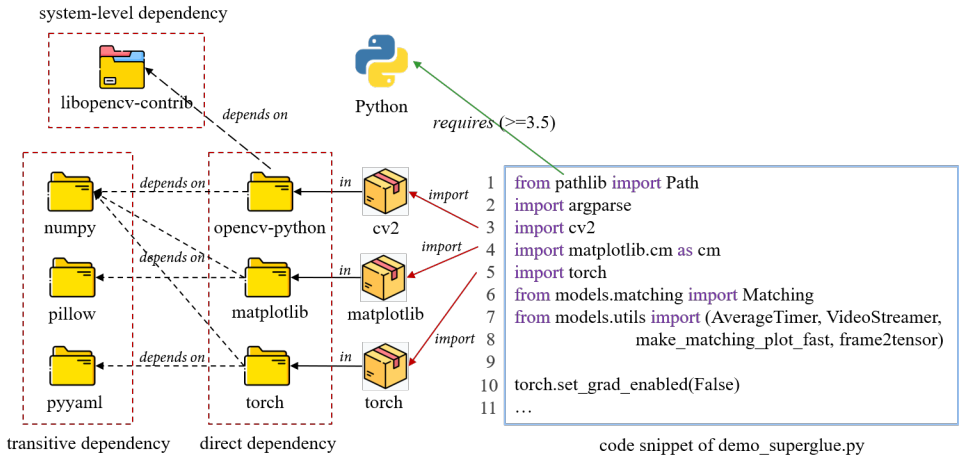

Figure 1: An example code snippet and its dependencies.

2.2 Motivating Example

Figure 1 shows a Python code snippet of a GitHub project, an implementation of SuperGlue network[1]. The code snippet belongs to `demo_superglue.py` (`superglue` for short), running SuperPoint and SuperGlue features matching an anchor image with live images [32]. Table 1 lists `superglue`'s feasible dependencies, where the expressions in brackets (e.g., ⩾ 3.5) are version constraints. `Superglue`'s dependencies include:

(1) *Third-party package.* Third-party packages are the basic installation units instead of their contained modules. Therefore, `torch`, `opencv-python`, and `matplotlib` are *direct dependencies* as they contain the directly used top-level modules `torch`, `cv2`, `matplotlib` (line 3-5), and the second-level module `cm`. Besides, `numpy`, `pillow`, `pyyaml` are *transitive dependencies* since the *direct dependencies* may use them.

(2) *System library.* Some third-party packages are partially implemented in C, and the C code may depend on system libraries. `libopencv-contrib` is a system-level dependency as the third-party package `opencv-python` depends on it.

(3) *The Python interpreter.* The Python interpreter is necessary for providing a Python environment and standard modules. `pathlib` and `argparse` (line 1-2) are *standard modules* installed with the Python interpreter, and `Path` is a second-level module in `pathlib`.

(4) *Local module.* A code snippet may use other modules implemented in a program itself, such as `models` (line 6-8) implemented in another file `models.py`. However, we do not regard local modules as they are self-contained resources.

Besides, the imported modules imply version constraints on such dependencies, including:

(1) *Python version constraint.* The code restricts the Python interpreter version must be 3.5 or later as the standard module `pathlib` is introduced since Python 3.5.

(2) *Third-party package version constraint.* `matplotlib 3.4.2` is the latest version, and it requires the transitive dependency `pillow` ⩾ 6.2.0. Moreover, `matplotlib`, `opencv-python` and `torch` depend on `numpy` simultaneously, and the latest versions of the first two packages restrict `numpy` later than 1.16 and 1.19.3, respectively. Thus, `numpy` ⩾ 1.19.3 is a version constraint restricting `numpy` compatible with all the other third-party packages using it.

Challenges. This example indicates that inferring dependencies for a Python program has several difficulties.

(1) A Python package includes at least one module, and the names of the package and the module can be different, which makes developers have to know to what packages the imported modules belong when reusing open-source code.

(2) System libraries are necessary *transitive dependencies* used by the imported modules, but few documents record the dependent relations between third-party packages and system libraries. The prior work finds missed system libraries by iteratively analyzing runtime error logs [24] with expensive costs.

(3) The differences in contained modules would restrict versions of a dependent third-party package. Therefore, additional efforts have to be taken to identify the compatible package versions.

(4) The imported standard modules and leveraged syntax features implicitly restrict compatible Python interpreter versions, but it is not easy to uncover such constraints in the target program code.

(5) Complex interdependent relations and dependency version constraints among a program and its dependencies may result in version conflicts [38]. Therefore, a dependency at the latest version may be incompatible [30], and in consequence, the latest compatible version of each dependency has to be figured out. In other words, we should follow the optimization principle of "using dependencies *as new as possible*" when multiple compatible versions exist because installing the latest versions follows `pip`'s working mechanism [15] and may fix bugs and security vulnerabilities in prior versions [20].

Limitations of the existing work. The state-of-the-art approaches, pipreqs [6], DockerizeMe [24] and SnifferDog [37] mainly focus on inferring third-party package dependencies. We execute pipreqs and DockerizeMe to infer dependencies for `superglue`.

Without regarding the version constraint of the Python interpreter, DockerizeMe takes Python2.7 as the default interpreter and fails with "`ImportError: No module named pathlib`." This is

[1]https://github.com/magicleap/SuperGluePretrainedNetwork

Table 1: Environment dependencies of the example code

Dependent Resource	Version
Python	3.9 (⩾ 3.5)
opencv-python	4.5.2.52
matplotlib	3.4.2
torch	1.8.1
numpy	1.20.3 (⩾ 1.19.3)
pillow	8.2.0 (⩾ 6.2.0)
libopencv-contrib3.2	3.2.0+dfsg-4ubuntu0.1

`Pyyaml` is not a necessary *transitive dependency* as the *direct dependency* `torch 1.8.1` no longer uses it

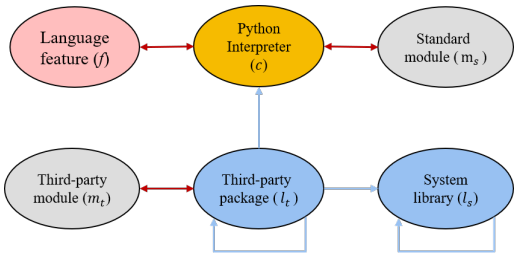

Figure 2: Knowledge graph model

because the standard module is introduced since Python 3.5. Although we use Python 3.5 or later, DockerizeMe still fails with another `ImportError` as it cannot identify the dependency between `opencv-python` and `libopencv-contrib3.2`.

Pipreqs directly uses the local Python environment and would also fail with the `ImportError` if the Python version is older than 3.5. Besides, pipreqs pays no attention to dependent system libraries, and hence it still fails even if we use a compatible Python version.

SnifferDog infers dependencies for Jupyter notebooks. Notably, we cannot successfully execute SnifferDog since its public repository [11] does not offer the critical component API-bank. Although SnifferDog analyzes modules, functions, and classes in third-party packages, its limitations are (1) not regarding Python version constraint and dependent system libraries and (2) simply taking the latest package versions containing used APIs as dependencies.

2.3 Our Approach Overview

As a result, much knowledge is required for dependency inferences. (1) Syntax features and standard modules of each Python interpreter version are required for identifying compatible Python versions. (2) Modules in each third-party package version should be known to determine what package versions are compatible. (3) Dependencies between Python packages and system libraries are required for installing dependent system libraries proactively. (4) Version constraints among third-party packages and between the Python interpreter and third-party packages are required to avoid incompatibilities and dependency conflicts (DCs) [38].

We are motivated to design a knowledge-based and constraint-aware technique to infer environment dependencies at the latest compatible versions. The technique comprises two main parts, i.e., a knowledge graph offering knowledge relevant to the three types of dependencies and a tool inferring Python program dependencies by solving version constraints.

Table 2: Symbols in dependency knowledge graph model

Symbol	Description
c	Python interpreter
l_t	Third-party package
l_s	System library
f	Language syntax feature
m_s	Standard module
m_t	Third-party module
$\rightarrow$	*depend-on* relationship
$\leftrightarrow$	*associated-with* relationship

In the first part, we propose a dependency knowledge graph model to portray the relations among third-party packages, the Python interpreter, system libraries, and other related entities. Then, we identify multiple sources from which we acquire data and information relating to the required knowledge. Finally, we extract knowledge from the data with several methods and construct a dependency knowledge graph PyKG.

In the second part, we design a technique PyEGo to infer dependencies for Python programs. Given a program, PyEGo extracts its syntax and module features and gets the program-related dependency candidates from PyKG. After that, we propose dependency constraints according to the dependency inference requirements, and PyEGo infers final dependencies from the candidates by solving dependency constraints.

3 KNOWLEDGE GRAPH CONSTRUCTION

We structure the required knowledge as a dependency knowledge graph whose model is illustrated in Figure 2.

Dependency Knowledge Graph is defined as $G =< V, E >$, where (1) $V = \{n_i | i \geqslant 0, \forall n_i \in \{f, c, l_t, l_s, m_s, m_t\}\}$ is a set of vertices relevant to dependencies, meaning a vertex can be the *python interpreter* (c), a *third-party package* (l_t), a *system library* (l_s), a *third-party module* (m_t), a *standard module* (m_s), and a *syntax feature* (f). (2) $E = \{e_j | j \geqslant 0, \forall e_j \in \{dep, asso\}$ is a set of edges of two relation types, i.e., *depend-on* and *associated-with* are represented by symbols '$\rightarrow$' and '$\leftrightarrow$', respectively. Table 2 lists the symbols used in this definition.

On the one hand, *depend-on* relations describe inter-dependencies among the three types of dependencies. In particular, a third-party package can depend on other packages, the Python interpreter and system libraries, and a system library may depend on other libraries. In addition, the dependent relations between such dependency versions also specify the version constraints among them.

On the other hand, *associated-with* relations describe features of third-party packages and the Python interpreter. Thus, the knowledge graph model characterizes a third-party package version with the third-party modules they contain. It also characterizes a specific Python version with its standard modules and the supported syntax features.

Table 3: Data sources of dependency knowledge

Source	Acquired Data & Information
PyPI [8]	Top 10,000 popular third-party packages at each version
Libraries.io [4]	Popularity measured by *SourceRanks* of third-party packages
APT [1]	Information of system libraries
Python docs [9]	Syntax features of Python versions
Python environments	Python versions and their standard modules

3.1 Data Source

Table 3 lists the sources from which we directly acquire data and information, where

- PyPI [8] is the world-class Python library repository hosting 300k+ third-party open-source packages, and from which we crawl the most popular ones and their inter-dependent relations;
- Libraries.io [4] provides *SourceRanks* that measure the popularity of open-source Python libraries;
- APT [1] is the official system software repository of `Debian` family operating systems provisioning system libraries;
- Python official website [10] offers Python interpreter versions, and from which we download and install Python at various versions. We analyze installed Python environments and extract the standard modules;
- Online Python documents [9] provide new features, user guidance, and other information on each Python version.

Note that we sample data instead of crawling all to make a trade-off between the expensive cost of exhaustively acquiring data and adequate verification and demonstration of our work. Table 3 also lists what data and information we acquire from the data sources.

3.2 Knowledge Extraction

We propose several methods for extracting knowledge, particularly syntax features, dependency version constraints of third-party packages, and dependencies between third-party packages and system libraries, from the obtained data.

3.2.1 Syntax feature extraction and representation. Syntax features supported by each Python interpreter version are scattered in online documents and usually described in natural language, and hence how to extract and present such knowledge is concerned.

We notice that the "*what's new*" document of each Python version explains new features compared with the previous version, such as "*What's New In Python 3.9*" [12]. Therefore, we extract *context-free* syntax features from the "*New Features*" section in each feature document because such features can be directly recognized in a target program without needing to analyze program contexts and other factors. For example, "*positional-only parameters*" (`PEP 570`) [27] is a context-free syntax feature can be recognized when a symbol "/" appears like a parameter in a function definition based on the static analysis. Conversely, "*Dictionary Merge & Update Operators*" (`PEP 584`) [33] is not a such feature as the context (i.e., operand types) must be concerned. Besides, other features, like "*New Parser*" (`PEP 617`) [22], irrelevant to source code are also discarded. In this way, we systematically investigate feature documents of six Python versions and recognize 26 context-free syntax features (syntax features for short hereafter). $c(v) \leftrightarrow \{f_1, f_2, \cdots\}$ denotes the syntax features supported by the Python interpreter at version v (or a version range).

Furthermore, to automate syntax feature recognition for a target Python program, we manually transform 19 out of the 26 extracted syntax features into regular expressions. For example, the knowledge of the feature "*positional-only parameters*" introduced since Python 3.8 is represented as:

$$c(\geqslant 3.8) \leftrightarrow \text{"}\texttt{def \textbackslash S*\textbackslash(.*, ?/.*\textbackslash)}\text{"}$$

Notably, we do not present all extracted syntax features due to space limitations, and they are available in our released knowledge graph.

3.2.2 Version constraint extraction and module identification. A third-party package's metadata usually provides information on contained third-party modules, compatible Python versions, and version constraints on its dependent packages. However, such information is scattered in several metadata files, and even worse, some information is incomplete or absent.

For a third-party package version $l_t(v)$, we extract its compatible Python versions and dependent third-Party packages by analyzing its metadata in the files `METADATA` and `requirements.txt`. The obtained information is represented as $l_t(v) \rightarrow c(v_c)$ and $l_t(v) \rightarrow \{l_{t1}(v1), l_{t2}(v2), \cdots\}$, where $c(v_c)$ is compatible Python versions and $\{l_{t1}(v1), l_{t2}(v2), \cdots\}$ is a set of dependent package versions.

The differences in contained third-party modules (particularly lower-level modules) are important for distinguishing versions of a package. However, the file `top_level.txt` in a package only records its top-level modules. Therefore, we traverse each package top-down to get its modules at each level. The information is represented as $l_t(v) \leftrightarrow \{m_{t1}, m_{t2}, \cdots\}$.

3.2.3 Dependency inference between third-party packages and system libraries. As aforementioned, dependencies between a package and its system libraries are absent. We mine such dependency knowledge with two methods, i.e., (a) association mining-based dependency inference and (b) similarity-based dependency inference. Since the former is similar to that proposed in prior work [24], we only elaborate on the latter below.

We observe that some *pip-installable* third-party packages have similar *apt-installable* distributions with slight differences in their names and structures. For example, `python-matplotlib`, an *apt-installable* package, is similar to the package `matplotlib` on `PyPI` as their names are similar and they both contain the same top-level modules `matplotlib`, `pylab`, and `mpl_toolkits`. Moreover, if an *apt-installable* package depends on a set of system libraries $\{l_{s1}, l_{s2}, \cdots\}$, the similar *pip-installable* one is likely depending on them, too. Thus, we can get dependent system libraries of an *apt-installable* package with the command "`apt-cache depends`". And then, we search for the similar *pip-installable* package (l_t) of the *apt-installable* one (l'_t) by measuring the comprehensive similarity (Equation 1) between their names (Equation 2) and contained top-level modules (Equation 3). l_t and l'_t are similar only if their

similarity exceeds a threshold set with 0.8 in practice. In Equation 2, $LCStr()$ returns the longest common substring of their names and $Max()$ returns the max length of their names. In Equation 3, $Module()$ returns the top-level module set of a package.

$$S(l_t, l'_t) = \frac{SN(l_t, l'_t) + SM(l_t, l'_t)}{2} \tag{1}$$

$$SN(l_t, l'_t) = \frac{|LCStr(l_t, l'_t)|}{Max(l_t, l'_t)} \tag{2}$$

$$SM(l_t, l'_t) = \frac{|Module(l_t) \cap Module(l'_t)|}{|Module(l_t) \cup Module(l'_t)|} \tag{3}$$

The inferred dependency knowledge is represented as $l_t \rightarrow \{l_{s1}, l_{s2}, \cdots\}$. For instance,

`opencv-python`$\rightarrow$ {`libopencv-contrib3.2`}

means "the *pip-installable* package `opencv-python` depends on the system library `libopencv-contrib3.2`."

3.3 Knowledge Graph Construction and Update

We organize extracted knowledge as a knowledge graph named PyKG, which is stored in *Neo4j* [36], a popular graph database. At the time of writing, PyKG contains about 256 thousand nodes and 1.9 million relations.

PyKG is extensible and evolvable. On the one hand, PyKG can update periodically as most data acquisition and knowledge extraction are automated. For each existed package, PyKG re-accesses PyPI and compares the local and the online versions. If the latest version is not fetched, PyKG automatically crawls it, analyzes its metadata, extracts corresponding knowledge, and makes a synchronization. In practice, PyKG synchronizes every three months, and each synchronization takes about 1.5 days. On the other hand, PyKG can increment in a similar way for crawling new packages and extracting relevant knowledge.

4 ENVIRONMENT DEPENDENCY INFERENCE

Figure 3 depicts PyEGo's workflow. The first step extracts imported modules and used syntax features of a target program via static analysis. Next, PyEGo gets dependency candidates by querying PyKG with extracted modules and syntax features. Finally, PyEGo outputs the inferred dependencies at the latest compatible versions based on constraint solving.

4.1 Program Feature Extraction

For a target program $\mathcal{P} = \{p_m | m \geqslant 1\}$ comprising a set of `.py` files, PyEGo extracts $\mathcal{P}$'s features, i.e., used syntax features, imported third-party modules and standard modules. For each python file $p_i (1 \leqslant i \leqslant m)$ in $\mathcal{P}$,

(1) PyEGo matches the code against each regular expression in PyKG that represents a syntax feature and groups identified syntax features in a set $S(p_i)$.
(2) PyEGo parses p_i into an abstract syntax tree (AST) and extracts its used modules from `import` statements. Next, in assistance with PyKG, PyEGo filters out local modules and groups the remaining ones in a standard module set ($M_s(p_i)$) and a third-party module set ($M_t(p_i)$), respectively.

In this way, PyEGo iteratively analyzes all `.py` files and integrates their features as $F(\mathcal{P}) = \{S, M_t, M_s\}$, where S, M_t and M_s denote $\mathcal{P}$'s syntax feature set, third-party module set, and standard module set, respectively.

For example, `superglue`'s (Sec. 2.2) feature set includes

- $S = \phi$, meaning no special syntax feature is identified,
- $M_s = \{argparse, pathlib, pathlib.Path\}$, and
- $M_t = \{cv2, torch, matplotlib, matplotlib.cm\}$.

4.2 Dependency Candidate Identification

PyEGo identifies $\mathcal{P}$'s dependency candidates by querying PyKG with $F(\mathcal{P})$.

Python interpreter candidates. PyKG returns all Python versions not only supporting all syntax features in $F(\mathcal{P}).S$ but also containing all standard modules in $F(\mathcal{P}).M_s$ as an interpreter candidate set $C(\mathcal{P}) = \{c^{v1}, c^{v2}, \cdots\}$. Thus, $C(\mathcal{P})$ implies Python interpreter version constraint of $\mathcal{P}$.

For example, the usage of standard module `pathlib` makes `superglue` compatible with Python 3.5 or later as the module is introduced since Python 3.5. Therefore, its candidate Python version set is $C(superglue) = \{c^v | 3.5 \leqslant v \leqslant 3.9\}$ (the latest Python version is 3.9).

Third-party package candidates. For each third-party module m_t in $F(\mathcal{P}).M_t$, PyKG returns all third-party package versions containing the module, i.e., $L_t(m_t)$. In this way, all possible third-party packages directly used in $\mathcal{P}$ are grouped in $L_t(\mathcal{P})$. On the other hand, PyEGo constructs $L'_t(\mathcal{P})$ that comprises all transitive dependent third-party packages of $\mathcal{P}$.

This step concerns the following case and heuristically filters the initial third-party package candidates. For a module simultaneously contained in several different third-party packages, we heuristically select the most popular package instead of all for the following reasons: (1) packages containing the same modules may conflict with one another [5]; and (2) taking all the packages as dependency candidates would increase the time cost of the subsequent dependency inference. For example, `cv2` is a module in `opencv-python`, `opencv-python-headless`, and `opencv-contrib-python`. Thus, we select `opencv-python` as the candidate due to its popularity (with *SourceRank* 18 in Libraries.io [4]).

System library candidates. PyKG returns system libraries on which any element in $L_t(\mathcal{P})$ or $L'_t(\mathcal{P})$ depends. PyEGo takes all returned system libraries as $\mathcal{P}$'s dependency candidates of system libraries, i.e., $L_s(\mathcal{P})$.

Target program centric dependency graph. $\mathcal{P}$'s dependency candidates form a subgraph of PyKG, $G' =< V', E' >$, where the vertices in V' are dependency versions in $C(\mathcal{P}) \cup L_t(\mathcal{P}) \cup L'_t(\mathcal{P}) \cup L_s(\mathcal{P})$; E' contains dependent relations among them. Without loss of generality, we denote a dependency n at version v as n^v, whose dependency candidates in G' is $dep(n^v) = \{\alpha, \beta, \cdots\}$.

Figure 4 shows a part of `superglue`'s dependency graph formed by its dependency candidates. For simplicity, each vertex is annotated with a symbol plus a number representing a specific version. For example, the dependencies of vertex `MA34` (i.e., `matplotlib 3.4.2`) are represented as $dep(MA34) = \{PY39, PL82, NP20\}$.

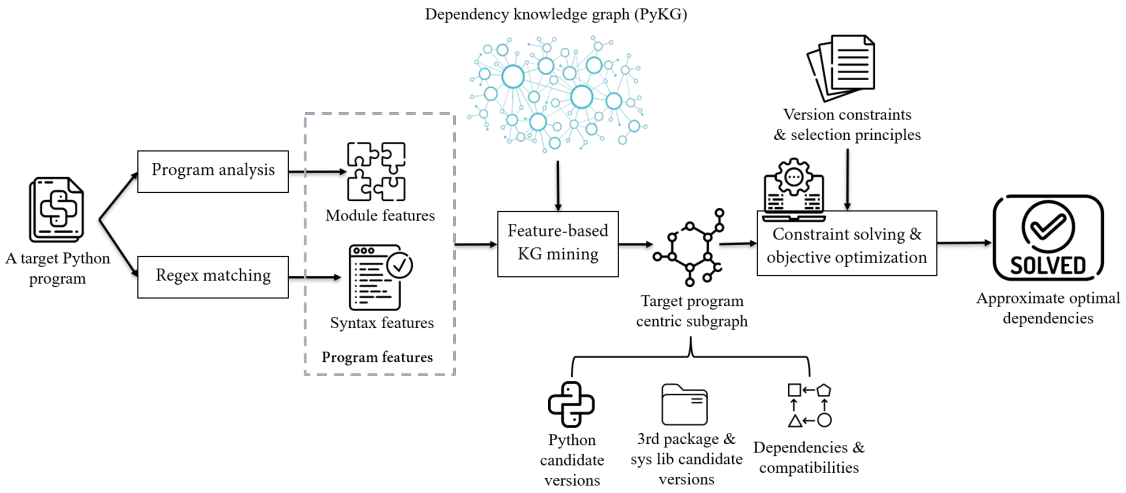

Figure 3: PyEGo's workflow of inferring environment dependencies for a Python program.

Symbol	Dependency
PY	Python
CV	Opencv-python
MA	Matplotlib
TH	Torch
NP	Numpy
PL	Pillow
YA	Pyyaml
LIB	Libopencv-contrib3.2

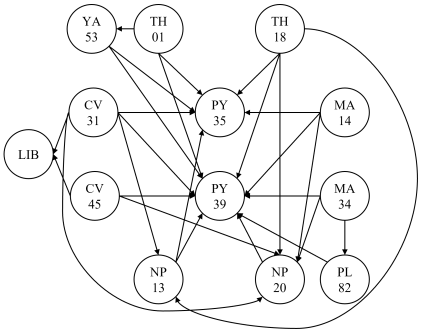

Figure 4: A part of Superglue's dependency graph.

4.3 Dependency Inference

As aforementioned, a package's latest version is preferred, but it is not always feasible due to DCs [38] and version constraint violations. Therefore, several requirements in inferring $\mathcal{P}$'s dependencies from the candidate set are summarized, i.e., the inferred dependencies must be compatible with one another, necessary, and as new as possible. To this end, PyEGo infers dependencies by solving dependency constraints with an optimization objective.

Dependency constraints. Suppose $\hat{G} =< \hat{V}, \hat{E} >, \hat{G} \subseteq G'$, is the graph formed by $\mathcal{P}$'s final inferred dependencies. It should satisfy the following constraints.

(1) *Existence constraint* restricts $\hat{G}$ must contain the Python interpreter and all $\mathcal{P}$'s directly dependent third-party packages, i.e.,

- $\forall n_i^{v1} \in C(\mathcal{P}) \cup L_t(\mathcal{P}), \exists n_j^{v2} \in \hat{G}.\hat{V}, n_i \equiv n_j$.

For example, in Figure 4, `Python`, `opencv-python`, `matplotlib`, and `torch` are dependencies must be installed. In contrast, `pyyaml` is not necessary if `torch 1.8.1` (TH18) is chosen.

(2) *Unique constraint* restricts $\hat{G}$ to contain only one version of each dependency n, i.e.,

- $\forall n_i^{v1}, n_j^{v2} \in \hat{G}.\hat{V}, n_i \neq n_j$.

For example, in Figure 4, one can choose either `torch 1.8.1` or `torch 0.1.2` (TH01), but cannot choose both at the same time.

(3) *Version constraint* restricts each dependency n^v in $\hat{G}$ must be compatible with all the other resources depending on it, i.e.,

- $\forall n_i^{v1} \in \hat{G}.\hat{V}, \forall n_j^{v2} \in dep(n_i^{v1}), \exists n_k^{v3} \in \hat{G}.\hat{V}, n_j \equiv n_k \wedge n_k^{v3} \in dep(n_i^{v1})$.

For example, in Figure 4, one cannot choose `matplotlib 3.4.2` and `Python 3.5` (PY35) at the same time, since `matplotlib 3.4.2` is not compatible with `Python 3.5`.

Optimization objective. We harmonize the latter two requirements above into an optimization objective. PyEGo sorts all versions of each dependency n in G' from the oldest to the latest as a vector $Ver(n) =< n^{v1}, n^{v2}, \cdots, n^{v0} >$. Notably, n^{v0} is a *virtual* version denoting no versions of n are selected. Therefore, the optimization objective is represented as Equation 4, where $index(n)$ denotes the index of the selected version in $Ver(n)$. For example, $index(n) = 0$ if the oldest version in $Ver(n)$ is selected. N is the set of all dependency candidates (not their versions) of $\mathcal{P}$.

$$O = max\Sigma_{\forall n \in N} index(n) \tag{4}$$

To maximize O, PyEGo first tries to select n^{v0} (i.e., not install any version of n), and if n is necessary or a constraint violation occurs, it replaces n^{v0} with the latest version of n satisfying all its relevant version constraints. To this end, PyEGo exploits the famous SMT solver Z3 [13] to obtain the optimized solution, i.e., the final inferred environment dependencies, such as the inference result of the example listed in Table 1.

5 EVALUATION

To evaluate the effectiveness and efficiency of PyEGo, we answer the follow research questions (RQs).

RQ3: How effective is PyEGo in inferring dependencies for Python programs?

RQ4: Does PyEGo outperform *state-of-the-art* approaches? How better is PyEGo than others?

RQ5: How efficient is PyEGo in inferring dependencies for Python programs?

For **RQ3**, PyEGo infers environment dependencies for Python programs in the dataset HG2.9K [24], 100 complex Python projects collected from Github, and 4,836 jupyter notebooks. We evaluate dependency inference results in terms of *accuracy* and *average inferred dependencies*.

For **RQ4**, we execute pipreqs [6] and DockerizeMe [24] with the datasets HG2.9K, the 100 collected projects and the 4,836 jupyter notebooks and compare their results with those of PyEGo. Notably, SnifferDog [37] is another related work, but we cannot replicate its experiment and quantitatively compare it with PyEGo as it is not executable due to the lack of critical component API-bank.

For **RQ5**, we compare PyEGo with pipreqs and DockerizeMe in terms of *average execution time*.

5.1 Methodology

5.1.1 Datasets. Three datasets are used in evaluations, i.e., HG2.9K [24], 100 open-source Python projects, and 4,836 jupyter notebooks.

HG2.9K contains 2,891 Python gists, i.e., single-file Python programs from GitHub gist service, experiencing *import* errors hard to fix in the prior work [23, 24].

In addition, following the criteria listed below, we create a dataset SD containing 100 real-world open-source Python projects collected from GitHub.

- *executable*, i.e., running without errors once all the dependencies are configured;
- *well documented*, i.e., document their dependencies explicitly;
- *popular*, i.e., having at least hundreds of stars;
- *diverse*, i.e., varying in forms and application types.

The projects in SD are third-party packages (47), applications (49) and tutorials (4), belonging to machine/deep learning (30), Internet (29), development (25), database (4), security (13), computer vision (14) and natural language processing (6) (a project can belong to multiple types). On average, these projects contain 58 Python files (with .py extensions), 10,821 lines of code (LOC), and 270 import statements.

Besides, we create a dataset JPD containing 4,836 jupyter notebooks. SnifferDog [37] provides a list of 6,004 notebooks in its repositories, and we try to download the notebooks and convert them into Python files. 1,168 notebooks fail in download or conversion, and we take the rest 4,836 notebooks as the final dataset. Notably, we comment out magic commands [2] contained in the notebooks as they cannot be executed by CPython interpreter.

5.1.2 Evaluation metrics. Equation 5 measures the effectiveness of dependency inference, where $Inf(programs)$ is the number of programs whose environment dependencies are successfully inferred within a dataset. Following the criterion in the prior work [24], an inference is considered successful only if the program does not encounter `ImportErrors` anymore. In addition, since the projects in SD record dependencies originally, we further compare their inference results with the documented dependencies to check whether the inferred environment dependencies are semantically correct.

$$accuracy = \frac{Inf(programs)}{total\ programs} \tag{5}$$

5.1.3 Experimental environment. Our experiments are performed on a computer running `Ubuntu 18.04 LTS` with 8-core 3.50 GHz CPU and 32 GB RAM.

5.2 Answer to RQ3

Experimental result. Table 4 shows the dependency inference results of PyEGo for programs in the three datasets.

(1) PyEGo successfully infers dependencies for 1,334 out of 2,891 (46.14%) gists in HG2.9K, 1.41 third-party packages and 2.31 system libraries per gist on average. Among the 1,334 gists,PyEGo infers Python 3.9 for 765 gists, Python 2.7 for 528 gists, and Python 3.8 for 41 gists, respectively.

(2) PyEGo successfully infers dependencies for 62 out of 100 (62%) projects in SD, 8.91 dependencies (4.01 third-party packages and 4.90 system libraries) per project on average. The accuracy on SD is significantly higher than that on HG2.9K since each project in SD is complete and executable originally. Therefore, they would not encounter `ImportErrors` caused by missing local modules. Among the 62 projects, PyEGo infers Python 3.9 for 51 projects, Python2.7 for 6 projects, and other Python versions for 5 projects. We inspect the inference results of the 62 projects and find that 4 results contain later dependency versions, but they do not change the target projects' execution semantics as the used APIs in the imported third-party package versions do not change. For example, jd-assistant [3] declares it depends on `pycryptodome 3.6.6`, while PyEGo infers version 3.10.1. We inspect the change log of `pycryptodome` [7] and confirm the used APIs, i.e., `Crypto.PublicKey.RSA` and `Crypto.Cipher.PKCS1_v1_5`, do not change since 3.6.1. Thus, we consider the inferred result is correct.

(3) PyEGo successfully infers dependencies for 2,945 out of 4,836 (60.90%) notebooks in JPD, 8.51 dependencies (3.30 third-party packages and 5.21 system libraries) per project on average. Among the 2,945 notebooks, PyEGo infers Python3.9 for 2,193 notebooks, Python2.7 for 479 gists and other Python versions for 273 notebooks.

Failure root causes analysis. We inspect programs failing with incorrect dependency inference results and analyze the root causes. Overall, the root causes are classified into several categories.

(1) *Missing dependencies.* First, some programs fail due to missing third-party packages. Take the code snippet below as an example, PyEGo infers `numpy 1.16.1` and `scipy 1.2.3`, but fails to figure out what package contains the module `sparsesvd` as the knowledge is absent.

```
1  import scipy.sparse
2  import numpy
3  import sparsesvd
```

Second, some programs fail to install third-party packages because of missing required system libraries. For example, installing `dbus-python` without the system library `libdbus-1-dev` leads to "`error: Package requirements(dubs-1 >= 1.8) were not met`." PyEGo fails to figure out on what system libraries the third-party package depends.

Third, some programs fail due to missing *extra* dependencies. Some third-party packages are installed optionally (`PEP 508`) [19], i.e., they would not be automatically installed unless asking for them explicitly. For example, `xlrd` is an *extra* dependency of `pandas`,

Table 4: Dependency inference results of PyEGo and two state-of-the-art approaches

	Tool	ACC	ADP	ATP	AT (sec.)
HG2.9k	PyEGo	**46.14%** (1334/2891)	3.68	**1.39**	**0.69**
	pipreqs	10.27%(297/2891)	**1.52**	1.52	2.18
	DockerizeMe	30.72%(888/2891)	7.32	5.99	13.45
SD	PyEGo	**62.00%**(62/100)	8.98	**4.01**	2.52
	pipreqs	45.00%(45/100)	**6.25**	6.25	**2.43**
	DockerizeMe	23.00%(23/100)	13.18	10.41	10.37
JPD	PyEGo	**60.90%**(2,945/4,836)	8.51	**3.30**	**2.86**
	pipreqs	52.92%(2,559/4,836)	**3.41**	3.41	3.11
	DockerizeMe	46.20%(2,234/4,836)	8.48	6.86	7.97

ACC is the dependency inference accuracy; ADP is the average inferred dependent packages, including system libraries and third-party packages; ATP is the average inferred dependent third-party packages; AT is the average execution time.

which would not be automatically installed with "`pip install pandas`". Executing function `pandas.read_excel()` without `xlrd`, the program would encounter an import error, i.e., "`ImportError: Missing optional dependency 'xlrd'...`".

(2) *Incompatible dependency versions.* First, some programs fail due to incompatible dependency versions induced by incorrect version constraints documented in third-party packages' metadata. For example, `pyarrow 2.0.0` specifies it is compatible with `Python >=3.5`. However, it is incompatible with `Python 3.9` and would fail with "`Installing build dependencies ... error.`".

Second, some programs fail due to incompatible classes or functions in third-party package versions. For example, the function `numpy.rank` is removed since `numpy 1.18`. Unfortunately, PyEGo cannot identify such an incompatible change as it focuses on features at module granularity.

(3) *Incompatible operating systems.* Some programs fail due to their dependent third-party packages are OS-specific. For example, `pyobjc-core` is only compatible with `OS X`, and installing it in Linux would fail with "`error: PyObjC requires macOS to build`."

(4) *Conditional dependency.* A program conditionally declares its dependencies on standard modules specific to `Python2` and `Python3` in `try-catch` or `if-else` code blocks for being compatible with both Python versions. In the code snippet below, `cPickle` and `pickle` are standard modules specific to `Python2` and `Python3`, respectively. PyEGo does not consider the `import` statements' context and cannot find a Python version containing both modules.

```
1 try:
2     import cPickle as pickle
3 except ImportError:
4     import pickle
```

5.3 Answer to RQ4

Table 4 lists all dependency inference results of the three approaches (including PyEGo). Since pipreqs and DockerizeMe do not consider Python versions, we config them with Python 2.7 for HG2.9k, following the original experimental setting of DockerizeMe, and config them with Python 3.9, the version most projects compatible with, for SD and JPD. Notably, our replicated evaluation on DockerizeMe with HG2.9K is slightly different from the original, i.e., it successfully infers dependencies for 888 instead of 892 gists. The reason is that some packages that DockerizeMe records have been removed from PyPI and cannot be downloaded and installed anymore. Overall, PyEGo reaches the highest accuracy on both datasets. In particular,

- PyEGo's accuracy is about 3.5x, 0.4x and 0.2x higher than that of pipreqs on the datasets HG2.9K, SD and JPD, respectively;
- PyEGo's accuracy is about 0.5x, 1.7x and 0.3x higher than that of DockerizeMe on the datasets HG2.9K, SD and JPD, respectively.

In comparison,

- DockerizeMe reaches the lowest accuracy on SD since it does not consider the hierarchical structure of a Python project and only analyzes .py files at the top-level.
- Pipreqs reaches the lowest accuracy on HG2.9k as it does not concern system libraries and Python version constraints.

The reasons of PyEGo outperforming the other approaches are summarized as follows.

Finer-grained module analysis. Pipreqs and DockerizeMe only focus on top-level modules, while PyEGo analyzes lower-level modules in addition. In comparison, lower-level modules are usually distinctive features for third-party package versions and Python versions. In the code snippet below, `urllib` exists in both Python2 and 3, but the second-level module `urllib.parse` is only in Python3. On the other hand, DockerizeMe and pipreqs infer the code snippet depends on `google-api-python-client` through the top-level module `oauth2client`. However, only versions between 1.0 and 1.2 of the package contain `oauth2client.appengine`. Consequently, DockerizeMe and pipreqs recommend the incompatible latest version 2.15, and conversely, PyEGo infers the package at version 1.2 and solves the `ImportError`.

```
1   import urllib.parse
2   from oauth2client import appengine
```

Discovering more dependent system libraries. Pipreqs does not consider system libraries, and DockerizeMe only discovers such dependencies based on association mining. In comparison, PyEGo discovers more dependencies with an additional similarity-based

Table 5: Pairwise comparison results

	# programs	ADP	ATP	AT (sec.)
HG2.9K-PyEGo	246	2.76	**1.37**	**0.46**
HG2.9K-pipreqs	246	**1.43**	1.43	1.11
HG2.9K-PyEGo	734	4.23	**1.63**	**0.49**
HG2.9K-DockerizeMe	734	**3.08**	2.94	5.10
SD-PyEGo	38	6.24	**2.92**	**0.55**
SD-pipreqs	38	**3.00**	3.00	1.21
SD-PyEGo	19	7.84	**3.26**	**0.53**
SD-DockerizeMe	19	**7.74**	6.95	8.78
JPD-PyEGo	2,287	6.85	**2.85**	**2.91**
JPD-pipreqs	2,287	**2.86**	2.86	2.93
JPD-PyEGo	2,035	6.32	**2.72**	**2.80**
JPD-DockerizeMe	2,035	**5.26**	4.82	6.18

The top two sub-datasets are gists in HG2.9K whose dependencies are inferred by both PyEGo and pipreqs, and both PyEGo and DockerizeMe. The middle two sub-datasets are projects in SD whose dependencies are inferred by both PyEGo and pipreqs, and both PyEGo and DockerizeMe. The bottom two sub-datasets are projects in JPD whose dependencies are inferred by both PyEGo and pipreqs, and both PyEGo and DockerizeMe.

approach (Sec.3.2). For instance, PyEGo discovers `av` depending on `libavcodec-dev`, and `pyldap` depending on `libldap2-dev` and `libsasl2-dev`. Such knowledge helps PyEGo achieve high accuracy.

Python version compatibility awareness. DockerizeMe takes Python2.7 as the default interpreter, and pipreqs only uses the local Python versions directly. However, not all programs are compatible with pre-installed Python versions. PyEGo analyzes syntax features and standard modules used in programs, and thus PyEGo recommends a compatible Python version for each target program.

Dependency conflicts prevention. Both pipreqs and DockerizeMe simply take the latest package versions without concerning dependency conflicts. PyEGo avoids potential DCs by solving constraints among dependencies. For example, in the following code snippet, the latest version of `torchmeta` and `torchvision` are 1.7.0 and 0.10.0, respectively. However, `torchmeta 1.7.0` depends on `torchvision<0.10.0`. Thus, installing the latest versions of both packages would result in a DC. Conversely, PyEGo installs `torchmeta 1.7.0` and `torchvision 0.9.0` and avoids the DC.

```
1  from torchvision import models
2  from torchmeta.utils.data import BatchMetaDataLoader
```

On average, PyEGo infers the fewest third-party packages (ATP) but a little more system libraries than pipreqs and DockerizeMe. On the one hand, although pipreqs does not consider system libraries, some programs happened to depend on system libraries installed in default. On the other hand, some obtained dependencies between system libraries and third-party packages are false positives, resulting in unnecessary system libraries. However, a little more false positives is an acceptable price to pay for the higher accuracy.

To make a more thorough evaluation, we perform a set of pairwise comparisons. That is, we compare PyEGo with every other approach based on the programs whose dependencies are successfully inferred by both. For example, we compare PyEGo with pipreqs based on the gists whose dependencies are successfully inferred by both of them. Table 5 lists the pairwise comparison results. We notice that PyEGo can infer dependencies for most programs whose dependencies are successfully inferred by pipreqs and DockerizeMe.

5.4 Answer to RQ5

We compare PyEGo with DockerizeMe and pipreqs in terms of their average execution time. As Table 4 shows, PyEGo runs 2.2x and 18.5x faster than pipreqs and DockerizeMe on HG2.9K, as well as 0.1x and 1.8x faster than pipreqs and DockerizeMe on JPD, respectively. PyEGo runs 3.1x faster than DockerizeMe, but almost equally to pipreqs on SD. However, unlike pipreqs, which immediately exits once encountering a `SyntaxError` (caused by incompatible Python versions) during program analysis, PyEGo tries the other Python version until it finds a compatible one. PyEGo runs faster on all sub-datasets in the pairwise comparison since pipreqs would no longer encounter any `SyntaxErrors` and exit immediately.

PyEGo avoids online queries by storing the obtained knowledge in PyKG locally, and thus, its performance bottleneck becomes querying knowledge graph PyKG. Hence, we partially cache PyKG (10% in practice) to speed up PyEGo. Take the dataset SD as an example, PyEGo takes 2.52s in a dependency inference on average with cache, 1.79x faster than without cache. In comparison, pipreqs only locally records modules in third-party packages and has to query PyPI for the latest version, increasing its time overhead. On the other hand, DockerizeMe runs slowly because it directly accesses its knowledge base step by step without leveraging cache.

6 DISCUSSION

We discuss some future work to deal with limitations.

More comprehensive knowledge acquirement. PyEGo is limited in the equipped domain knowledge, e.g., PyKG stores only a small part (e.g., 1/30 third-party packages in PyPI) of domain knowledge. Thus, PyEGo does not know to what packages the unrecognized imported modules belong. Therefore, we will continuously and incrementally enrich PyKG by acquiring more domain knowledge.

OS-specific dependency identification. Our failure root cause analysis found that some third-party packages are compatible with specific OSes, e.g., macOS. Currently, PyEGo concentrates on dependencies in Linux, and we will enrich PyKG with OS-level compatibility information in the future.

Finer-grained feature utilization. Functions and classes are also important for distinguishing versions of the Python interpreter and packages, especially in situations where two versions of a package only differ in implementation details. However, PyEGo only concerns module-level features and cannot recognize the function-

and class-level differences. We will extract and utilize such finer-grained features in standard and third-party modules for improving dependency inference accuracy.

Conditional dependency identification. PyEGo fails to infer *conditional dependencies* for target programs. We will explore context-aware program analysis to identify such dependencies.

7 THREATS TO VALIDITY

Internal validity threats come from the acquired knowledge, particularly the dependencies between third-party packages and system libraries. The similarity- and association-based dependency mining may introduce false positives (Sec. 3.2). We mitigate the threat by (1) leveraging a comprehensive metric for increasing the accuracy of similarity-based mining and (2) setting the threshold with a moderately high value to filter out more false positives.

The external validity concerns the generality of our work. We evaluate PyEGo by resolving `ImportErrors` for 2,891 single-file Python programs. In particular, each program in HG2.9K is a "*hard gist*" whose `ImportErrors` cannot be resolved by the naive algorithm [24]. The experimental result reveals that PyEGo is more effective than state-of-the-art approaches. On the other hand, we evaluate PyEGo on 100 real-world projects selected from GitHub based on several criteria and 4,836 jupyter notebooks. These projects are more complex with multiple code files. The experimental result shows that PyEGo is scalable to infer dependencies for complex Python projects of various types, and most inferred dependencies do not change execution semantics.

Construct validity refers to the suitability of our evaluation measures. Like the prior study [24], we think a dependency inference is successful only if the target program no longer experiences `ImportErrors`. We evaluate the dependency inference result of a Python project more strictly. A dependency inference of a complete Python project is successful if the project can execute correctly and the inference is consistent with its originally declared dependencies.

8 RELATED WORK

Dependency inference. The most relevant work is DockerizeMe [24], SnifferDog [37], and pipreqs [6]. DockerizeMe infers third-party package and system library dependencies using a combination of static analysis, dynamic analysis, and association rule mining. Pipreqs, a popular open-source tool, generates a `requirements.txt` file for a Python program based on `import` statements analysis. It resolves the inconsistencies between package names and module names based on a pre-constructed dictionary and queries PyPI [8] on the fly. SnifferDog [37] analyzes Jupyter notebooks to determine candidates for required packages and versions based on a database of APIs. PyEGo is more effective since it (1) concerns more dependencies, (2) considers version constraints among dependencies, and (3) is configured with rich and detailed dependency knowledge.

Dependency conflict management. WatchMan [38] detects DCs in the PyPI ecosystem. Riddle [40] generates tests to collect crashing stack traces to facilitate DC issue diagnosis of Java projects based on the empirical study findings [39]. Pradel et al. [31] proposed a detection strategy for DCs between JavaScript libraries. LibHarmo [26] detects library version inconsistencies for Java Maven projects and interactively suggests a harmonized version with the least harmonization efforts. Sensor [41] synthesizes test cases to trigger inconsistent behaviors of the APIs with the same signatures in conflicting Java library versions.

Breaking change detection. PyDFix [30] detects and fixes unreproducibility in Python builds caused by breaking changes of dependencies. V2 [25] detects breaking changes based on Python program crash information. It fixes crashes by repeatedly building environments and running programs with inferred dependencies in a *trial-and-error* manner, inducing intolerable time overhead. Mezzetti et al. [28] presented a technique, type regression testing, to detect breaking changes in Node.js libraries. Through cross-project testing and analysis, DeBBI [17] detects backward behavioral incompatibilities between Java software libraries and client software projects. Mujahid et al.[29] leveraged automated test suites of other projects depending upon the same dependencies to test newly released *npm* package versions.

Python ecosystem study. Valiev et al. [34] performed a mixed-methods study on ecosystem-level factors affecting the sustainability of open-source Python projects. Bommarito et al. [16] and Chen et al. [18] conducted empirical studies on PyPI and language features, respectively. Vu et al. [35] found that PyPI is an attractive target for attackers to trick developers into using malicious packages. They studied the attacks and proposed an approach to identify combosquatting and typosquatting [21] packages automatically.

9 CONCLUSION

Configuring a Python program execution environment is non-trivial due to complex dependencies. This work proposes an automated dependency inference technique PyEGo. Assisted with a dependency knowledge graph, PyEGo considers dependencies of third-party packages, the Python interpreter and system libraries, and infers dependencies for a program by constructing its dependency graph and solving constraints in it. The evaluation shows that PyEGo is more effective and efficient than the state-of-the-art approaches. In the future, we plan to enhance PyEGo in several aspects, e.g., considering finer-granularity features, improving dependency mining between system libraries and third-party packages, automating language feature identification and representation.

10 DATA AVAILABILITY

PyEGo and its experimental data are publicly available at https://github.com/PyEGo/PyEGo.

ACKNOWLEDGEMENTS

This work was partially supported by National Key R&D Program of China (2017YFA0700603), National Natural Science Foundation of China (61732019, U20A6003, 62072444), Foundation of Science and Technology on Parallel and Distributed Processing Laboratory (61421102000402), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142).

REFERENCES

[1] [n.d.]. The Advanced packaging tool, or APT (from Wikipedia). Retrieved August 20, 2021 from https://en.wikipedia.org/wiki/APT_(software)
[2] [n.d.]. Built-in magic commands. Retrieved August 11, 2021 from https://ipython.readthedocs.io/en/stable/interactive/magics.html

[3] [n.d.]. Jd-assistant. Retrieved August 24, 2021 from https://github.com/tychxn/jd-assistant
[4] [n.d.]. Libraries.io. Retrieved August 10, 2021 from https://libraries.io/
[5] [n.d.]. Opencv-python. Retrieved September 2, 2021 from https://pypi.org/project/opencv-python/
[6] [n.d.]. pipreqs. Retrieved August 4, 2021 from https://github.com/bndr/pipreqs
[7] [n.d.]. Pycryptodome changelog. Retrieved August 24, 2021 from https://pycryptodome.readthedocs.io/en/latest/src/changelog.html#
[8] [n.d.]. PyPI. Retrieved August 7, 2021 from https://pypi.org/
[9] [n.d.]. Python Docs by version. Retrieved August 11, 2021 from https://docs.python.org/3/
[10] [n.d.]. Python Official Website. Retrieved August 13, 2021 from https://www.python.org/
[11] [n.d.]. SnifferDog repository. Retrieved August 24, 2021 from https://github.com/SMAT-Lab/SnifferDog
[12] [n.d.]. What's New in Python. Retrieved August 25, 2021 from https://docs.python.org/3.9/whatsnew/
[13] [n.d.]. Z3. Retrieved August 17, 2021 from https://github.com/Z3Prover/z3/wiki#background
[14] 2019. Customized dependency resolution / full required graph #3118. Retrieved July 30, 2021 from https://github.com/pypa/pipenv/issues/3118
[15] 2019. pip documentation v21.0.1. Retrieved July 30, 2021 from https://pip.pypa.io/en/stable/reference/pip_install/
[16] Ethan Bommarito and Michael Bommarito. 2019. An empirical analysis of the python package index (PyPI). *arXiv preprint arXiv:1907.11073* (2019).
[17] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124.
[18] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. An Empirical Study on Dynamic Typing Related Practices in Python Systems. In *Proceedings of the 28th International Conference on Program Comprehension*. 83–93.
[19] Robert Collins. 2015. Dependency specification for Python Software Packages. Retrieved July 30, 2021 from https://www.python.org/dev/peps/pep-0508/#extras
[20] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 349–359.
[21] John E Dunn. 2017. PyPI Python repository hit by typosquatting sneak attack. Retrieved July 28, 2021 from https://nakedsecurity.sophos.com/2017/09/19/pypi-python-repository-hit-by-typosquatting-sneak-attack/
[22] Lysandros Nikolaou Guido van Rossum, Pablo Galindo. 2020. New PEG parser for CPython. Retrieved August 11, 2021 from https://www.python.org/dev/peps/pep-0617/#overview
[23] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the executability of python code snippets on github. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 217–227.
[24] Eric Horton and Chris Parnin. 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 328–338.
[25] Eric Horton and Chris Parnin. 2019. V2: fast detection of configuration drift in Python. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 477–488.
[26] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, effort-aware library version harmonization. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 518–529.
[27] Mario Corchero Larry Hastings, Pablo Galindo and Eric N.Vander Weele. 2018. Python Positional-Only Parameters. Retrieved August 23, 2021 from https://www.python.org/dev/peps/pep-0570/
[28] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[29] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 466–476.
[30] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 439–451.
[31] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. 741–751.
[32] Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. 2020. Superglue: Learning feature matching with graph neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 4938–4947.
[33] Brandt Bucher Steven D'Aprano. 2019. Add Union Operators To dict. Retrieved August 11, 2021 from https://www.python.org/dev/peps/pep-0584/#abstract
[34] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 644–655.
[35] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and Combosquatting Attacks on the Python Ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 509–514.
[36] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. 2015. *Neo4j in action*. Vol. 22. Manning Shelter Island.
[37] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1622–1633.
[38] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: monitoring dependency conflicts for Python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 125–135.
[39] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 319–330.
[40] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I have a stack trace to examine the dependency conflict issue?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 572–583.
[41] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhi-liang Zhu. 2021. Will Dependency Conflicts Affect My Program's Semantics. *IEEE Transactions on Software Engineering* (2021).