**Knowledge-Based Environment Dependency Inference for Python Programs**

**Introduction**

Python programs often fail to run due to missing or incompatible dependencies. While existing tools mainly focus on third-party packages, they overlook dependencies on specific Python interpreter versions and system libraries. This paper introduces PyEGo, a technique that infers these dependencies using a knowledge graph (PyKG) to capture the relationships among different dependencies.

**Key Contributions**

1. **Dependency Knowledge Graph (PyKG)**: Captures relationships and constraints among Python interpreters, third-party packages, and system libraries.
2. **Automated Dependency Inference (PyEGo)**: Uses PyKG to infer the complete set of dependencies needed by Python programs.
3. **Evaluation**: Demonstrates PyEGo's higher accuracy compared to existing methods using multiple datasets.

**Datasets**

1. **HG2.9K**: 2,891 single-file Python programs (gists) from GitHub Gist with ImportErrors.
2. **SD (Selected Dataset)**: 100 open-source Python projects from GitHub.
3. **JPD (Jupyter Notebook Dataset)**: 4,836 Jupyter notebooks converted to Python files.

**Existing Approaches**

1. **pipreqs**: Generates a **requirements.txt** file based on import statements but ignores system libraries and Python interpreter versions.
2. **DockerizeMe**: Uses static and dynamic analysis to infer dependencies, focusing mainly on third-party packages.
3. **SnifferDog**: Analyzes Jupyter notebooks to determine required packages but ignores system libraries and interpreter versions.

**Knowledge Graph Model**

* **Vertices (V)**: Represent Python interpreters, third-party packages, system libraries, syntax features, standard modules, and third-party modules.
* **Edges (E)**: Represent dependency (**→**) and association (**↔**) relationships among vertices.

**Knowledge Extraction from Data Sources**

1. **Syntax Feature Extraction**:
   * **Source**: Python documentation.
   * **Method**: Extract context-free syntax features and transform them into regular expressions.
   * **Example**: Positional-only parameters (**c(≥ 3.8) ↔ "def \S\*\(.\*, ?/.\*\)"**).
2. **Version Constraint Extraction and Module Identification**:
   * **Source**: Package metadata files.
   * **Method**: Extract information about package dependencies and compatible Python versions.
3. **Dependency Inference Between Third-Party Packages and System Libraries**:
   * **Association Mining-Based Dependency Inference**: Uses statistical associations to infer dependencies.
   * **Similarity-Based Dependency Inference**: Identifies dependencies based on name and structural similarities between pip-installable and apt-installable packages.

**Environment Dependency Inference**

1. **Program Feature Extraction**:
   * **Static Analysis**: Extract syntax features, third-party modules, and standard modules from the target Python program.
   * **Example**: For a program importing **cv2** from **opencv-python**, **torch**, and using standard modules **argparse** and **pathlib**:
     + **Syntax Features**: None identified.
     + **Standard Modules**: **argparse**, **pathlib**.
     + **Third-Party Modules**: **cv2**, **torch**.
2. **Dependency Candidate Identification**:
   * **Python Interpreters**: Identify all Python versions that support the extracted features.
   * **Third-Party Packages**: Identify all versions of the imported third-party modules.
   * **System Libraries**: Identify system libraries required by the candidate third-party packages.
3. **Constructing the Dependency Graph**:
   * **Vertices**: Represent dependency candidates.
   * **Edges**: Represent dependency relations.
   * **Subgraph**: Formed by combining relevant vertices and edges from PyKG.
4. **Solving Constraints for Dependency Inference**:
   * **Existence Constraint**: Ensure the graph includes the Python interpreter and directly dependent packages.
   * **Unique Constraint**: Ensure only one version of each dependency is selected.
   * **Version Constraint**: Ensure compatibility among selected versions.
   * **Optimization Objective**: Select the latest compatible versions.
   * **SMT Solver (Z3)**: Use Z3 to solve the constraints and optimize the selection of dependency versions.

**Example**: For a program requiring **opencv-python** and **torch**:

* **Python Version Candidates**: Python 3.5 to 3.9.
* **Third-Party Packages**: **opencv-python 4.5.2.52**, **torch 1.8.1**.
* **System Libraries**: **libopencv-contrib** for **opencv-python**.

### **Detailed Steps for Constraint Generation and SMT Encoding Evaluation**

The evaluation section of the paper "Knowledge-Based Environment Dependency Inference for Python Programs" assesses the effectiveness and accuracy of PyEGo in inferring the complete set of dependencies for Python programs. The evaluation is carried out using three datasets and compares PyEGo's performance against existing approaches: pipreqs, DockerizeMe, and SnifferDog.

**Datasets Used for Evaluation**

1. **HG2.9K**:
   * **Description**: Contains 2,891 single-file Python programs (gists) from GitHub Gist known to encounter ImportErrors that are difficult to fix.
   * **Purpose**: Provides a benchmark for testing PyEGo's ability to resolve ImportErrors by inferring missing dependencies.
2. **SD (Selected Dataset)**:
   * **Description**: Includes 100 real-world open-source Python projects from GitHub. These projects are executable, well-documented, popular, and diverse in their application domains.
   * **Purpose**: Tests PyEGo's effectiveness on more complex and varied Python projects.
3. **JPD (Jupyter Notebook Dataset)**:
   * **Description**: Comprises 4,836 Jupyter notebooks converted into Python files.
   * **Purpose**: Evaluates PyEGo's performance on code written in Jupyter notebooks, which often contain unique dependencies.

**Evaluation Metrics**

The evaluation focuses on several key metrics to compare PyEGo with other tools:

* **Accuracy**: Measures the correctness of the inferred dependencies.
* **Average Inferred Dependencies**: Counts the average number of third-party packages and system libraries inferred per program.
* **Execution Time**: Assesses the time efficiency of PyEGo compared to other tools.

**Results**

1. **Accuracy**:
   * **HG2.9K**:
     + PyEGo achieved 46.14% accuracy, significantly higher than pipreqs and DockerizeMe.
     + pipreqs: 13.20%
     + DockerizeMe: 22.30%
   * **SD**:
     + PyEGo achieved 62% accuracy, outperforming pipreqs and DockerizeMe.
     + pipreqs: 24.70%
     + DockerizeMe: 36.10%
   * **JPD**:
     + PyEGo achieved 60.90% accuracy, higher than pipreqs and DockerizeMe.
     + pipreqs: 20.50%
     + DockerizeMe: 30.30%
2. **Average Inferred Dependencies**:
   * **HG2.9K**:
     + PyEGo inferred an average of 1.39 third-party packages and 2.31 system libraries per gist.
   * **SD**:
     + PyEGo inferred an average of 4.01 third-party packages and 4.90 system libraries per project.
   * **JPD**:
     + PyEGo inferred an average of 3.30 third-party packages and 5.21 system libraries per notebook.
3. **Execution Time**:
   * PyEGo demonstrated faster execution times compared to DockerizeMe, being comparable or slightly faster than pipreqs.
   * The evaluation shows that PyEGo is efficient in terms of execution time, making it suitable for practical use in real-world projects.