AutoPyDep: A Recommendation System for Python Dependency Management Utilizing Graph-Based Analytics

Dibyendu Brinto Bose
Virginia Tech
USA
brintodibyendu@vt.edu

Matthew Trimble
Virginia Tech
USA
mtrimble23@vt.edu

Travis Chan
Virginia Tech
USA
tchan89@vt.edu

Chris Brown
Virginia Tech
USA
dcbrown@vt.edu

Abstract

Managing software dependencies is increasingly challenging due to the complexity of modern development, often resulting in "dependency hell" with version conflicts, build failures, and runtime errors. To address these issues, we present *AutoPyDep*, a recommendation system for Python library dependency management. *AutoPyDep* features dependency analysis, relationship mapping, and predictive modeling for release categories and dates. By transforming release notes from 23 Python libraries into a graph network, we leverage NLP techniques and a community-based deepWalk algorithm to generate embeddings for tasks such as release category prediction and release date forecasting. Key contributions include a voting classifier achieving a robust F1 score of 0.8 and an ARIMA model with a Mean Absolute Error (MAE) of 1.8 months. *AutoPyDep* enhances dependency management accuracy, offering actionable insights for developers and supporting improved decision-making in software development. A demonstration of our tool is shared in this link.

CCS Concepts

• **Software and its engineering → Software libraries and repositories**.

Keywords

dependency management, recommendation system

ACM Reference Format:
Dibyendu Brinto Bose, Matthew Trimble, Travis Chan, and Chris Brown. 2025. AutoPyDep: A Recommendation System for Python Dependency Management Utilizing Graph-Based Analytics. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728576

1 Introduction

Managing software dependencies is a critical yet challenging task in modern software development. Issues such as version conflicts and dependency chains lead to the infamous "dependency hell" [3], or frustrations from dependency issues, causing build failures and runtime errors [8, 17]. These challenges are compounded by the NP-complete nature of dependency resolution [1], underscoring the need for more intelligent and effective solutions.

For instance, Python dependency management and corresponding tools are a "dumpster fire" [7]. Existing tools like `pip`[1] primarily focus on installation and version control, leaving gaps in dependency analysis and prediction [6] as these information are crucial in dependency decisions. Such limitations in detecting and preventing dependency errors can lead to costly delays and project failures [2, 4, 13]. Dependency management is also problematic across domains, including machine learning and blockchain [12, 20].

To address these gaps, we introduce *AutoPyDep*, a graph-based recommendation tool for Python dependency management. Unlike traditional tools, *AutoPyDep* predicts release categories and dates using machine learning, analyzes interdependencies, and provides actionable insights. Our approach transforms release note data from 23 Python libraries into a graph network, employing community detection, centrality analysis, and predictive models to support informed decision-making. Figure 1 illustrates our workflow, from data collection to actionable recommendations.EA

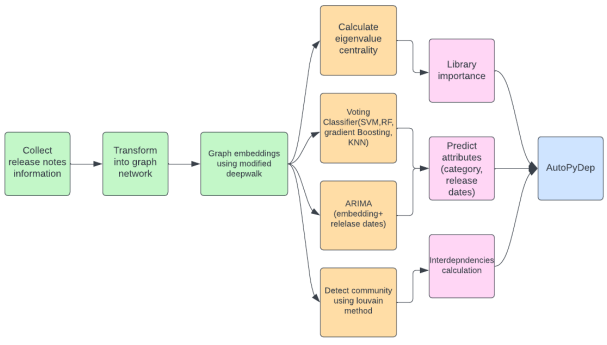


Figure 1: Overall workflow of our work

Our contributions include:

- A dataset of Python release notes capturing comprehensive details about release dates, categories, and versions.

[1] https://pypi.org/project/pip/

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728576

- A novel graph-based representation of library dependencies, enabling advanced analysis and predictions.
- The *AutoPyDep* tool, offering dependency analysis, relationship mapping, and predictive insights.

The predictive models in *AutoPyDep* achieve an F1-score of 0.8 for release category prediction and a Mean Absolute Error (MAE) of 1.8 months for release date forecasting. The tool and dataset are publicly available at https://anonymous.4open.science/r/Graph_Analysis-9D03/.

2 Related Work

2.1 Dependency Management

Dependency management often involves challenges such as dependency hell, caused by incompatible library versions, and is inherently NP-complete [1, 18]. Traditional approaches like SAT-based dependency solving and tools such as VeriBuild [9] utilize unified dependency graphs to detect issues efficiently [8]. *AutoPyDep* advances these efforts by combining graph-based models and NLP to dynamically predict and manage dependencies.

2.2 Recommendation Systems in Software Engineering

Recommendation systems in software engineering (RSSE) improve developer productivity and reduce cognitive load [10]. Techniques such as collaborative filtering [14], design pattern recommendation [15], and API suggestion systems [19] show practical utility. Prior studies highlight the importance of recommendation design in developer adoption [5, 11]. *AutoPyDep* builds on these approaches by integrating predictive analytics [16], providing recommendations for dependency updates for managing library versions.

3 AutoPyDep

3.1 Data Collection and Preprocessing

To address the lack of preexisting datasets for Python release notes, we curated a dataset by scraping information from GitHub and official library websites[2], [3] for 23 Python libraries. The collected data includes library names, versions, release notes, categories, and release dates, forming the foundation for our analysis and predictive models.

Preprocessing steps were applied to clean and standardize the release notes. Text was converted to lowercase, with special care taken to preserve version numbers and method calls. Tokenization split the text into individual words, while proper nouns and irrelevant standalone numbers were removed. Stop words were eliminated, but version numbers and method calls were retained for context. The cleaned tokens were rejoined into a single string after removing unnecessary characters, excessive spaces, and newlines. This robust preprocessing pipeline ensured the data's suitability for subsequent analysis and modeling.

3.2 Graph Creation

We transformed the processed dataset into a graph network following Algorithm 16. Each node in the graph represents a specific library version, and edges capture semantic and evolutionary relationships between them.

To construct the graph, we first calculated Term Frequency-Inverse Document Frequency (TF-IDF) and applied Named Entity (NE) to identify significant terms and entities within release notes. Dimensionality reduction using Truncated Singular Value Decomposition (SVD) highlighted key features, ensuring the graph's focus on relevant information. Edges were created based on cosine similarity between release notes, with a threshold of 0.4, determined through experimentation to balance relevance and graph density. Additionally, consecutive versions of the same library were explicitly connected to maintain version continuity.

Algorithm 1: Graph Creation from Release Notes

Input: Release notes dataset D
Output: Graph $G = (V, E)$
$V \leftarrow$ nodes representing each library version in D;
$E \leftarrow \emptyset$;
Calculate TF-IDF and apply NE on each release note in D;
Apply TruncatedSVD on TF-IDF+NE feature set to reduce dimensions;
foreach *pair of release notes* $(r_i, r_j) \in D$ **do**
  Calculate cosine similarity $sim(r_i, r_j)$;
  if $sim(r_i, r_j) > 0.4$ **then**
    Add edge (r_i, r_j) to E;
  end
end
foreach *library* L *in* D **do**
  foreach *consecutive version* (v_k, v_{k+1}) *of* L **do**
    Add edge (v_k, v_{k+1}) to E;
  end
end
return $G = (V, E)$;

This methodology produced a graph network where nodes represent library versions and edges represent their semantic and evolutionary connections, enabling downstream tasks such as dependency analysis and prediction.

3.3 Modified deepWalk

To generate meaningful network embeddings, we developed a modified version of the deepWalk algorithm, referred to as 'Mod2 (Community + Eigenvalue)'. This approach integrates Eigenvector Centrality to prioritize influential nodes and employs community detection to focus on densely connected subgraphs, enhancing the representation of interdependencies.

Traditional deepWalk techniques often emphasize local structures, resulting in scattered clusters and less effective capture of higher-order relationships. Our method addresses this by starting random walks from nodes with high Eigenvector Centrality scores, ensuring influential nodes are emphasized. Community detection further enhances the approach by directing walks through densely connected regions, capturing both local and global structures effectively.

[2] https://pytorch.org/
[3] https://www.tensorflow.org/

The Eigenvector Centrality for a node v in a graph $G = (V, E)$ is calculated as:

$$c_i(v) = \mathbf{v}_i^T \mathbf{e} \tag{1}$$

where $\mathbf{v}_i$ is the eigenvector of the adjacency matrix A corresponding to its largest eigenvalue λ_i, and $\mathbf{e}$ is a vector of ones. This measure ensures that the most influential nodes are selected as starting points for random walks, significantly improving embedding quality.

The embeddings generated using Mod2 demonstrate distinct clusters, indicating meaningful relationships and an effective representation of interdependencies within the graph. These embeddings serve as a robust foundation for downstream predictive tasks. (See Figure **??** for TSNE visualizations of the embeddings.)

3.4 Predicting Downstream Tasks

3.4.1 Library Importance. We evaluated the significance of updating library versions using Eigenvector Centrality. This metric identifies nodes that are highly connected and linked to other influential nodes, making it ideal for prioritizing critical libraries within the dependency network. Libraries with high centrality scores were deemed essential for timely updates, as they significantly influence overall system stability and functionality.

3.4.2 Interdependency Measurement. To analyze interdependencies among libraries, we employed the Louvain community detection algorithm. This method identifies clusters of libraries with dense internal connections, optimizing modularity to highlight meaningful relationships. By visualizing these clusters, developers can better understand the broader impact of updating specific libraries, aiding in effective dependency management.

3.4.3 Predicting Release Categories. To classify release notes into categories (e.g., Bug Fix, New Feature, Performance Improvement, and Security), we experimented with multiple classifiers, including SVM, Random Forest, Gradient Boosting, and K-Nearest Neighbors. A correlation analysis of the classifiers' predictions informed the selection of complementary models, which were combined using a Stacking Ensemble technique(see table **??** in appendix).

This ensemble, integrating SVM with Random Forest and Gradient Boosting, improved classification accuracy by leveraging the strengths of individual models. Hyperparameter tuning ensured optimal configuration, and 10-fold cross-validation demonstrated robust performance with a high macro F1 score across categories. This method provides reliable insights into the nature of library updates, supporting proactive dependency management.

3.4.4 Predicting Release Dates. For release date prediction, we combined embeddings with time-series modeling. Initial estimates were derived using a linear regression model trained on release intervals. These estimates were refined using an AutoRegressive Integrated Moving Average (ARIMA) model, leveraging historical release data. By averaging predictions from both models, we achieved accurate release date forecasts with a Mean Absolute Error (MAE) of 1.8 months. This hybrid approach effectively incorporates both textual and temporal information for robust predictions.

4 AutoPyDep Features Description

We developed *AutoPyDep* using Python, leveraging the CustomTkinter[4] library to implement a user-friendly desktop interface. Screenshots are provided with the tool walkthrough in Appendix **??**. Below, we describe the key features of the tool and their functionalities.

4.1 Home Tab

The *Home* tab serves as the starting point for *AutoPyDep*, providing buttons for loading input files (*Load File & Generate Recommendations*), saving outputs (*Save Recommendations*), and exiting the application (*Exit*).

4.2 Centrality Analysis

The *Centrality Analysis* tab highlights the importance of libraries in the dependency network. It displays input libraries, their versions, node centrality scores, and centrality concern levels, with color-coded rows to emphasize urgency (red for critical, green for non-critical). The tab also provides pip installation commands, which users can copy with a single click, facilitating library updates. Terminology definitions are included to enhance user understanding of centrality concepts.

4.3 Community Impact

The *Community Impact* tab provides insights into library interdependencies, listing the input libraries, their versions, and the number of other libraries they impact. Users can modify library versions to dynamically view the updated impact on the dependency network.

This tab also includes the *View Impact Graph* feature, which visualizes direct connections between libraries. Users can save this visualization for offline reference. Drop-down menus and scrollable frames ensure ease of navigation, regardless of the number of libraries or interdependencies.

4.4 Prediction Information

The *Prediction* tab presents predicted release categories and dates in a clear, tabular format. Release dates are displayed in a year-month format, helping developers anticipate updates and make proactive decisions about managing dependencies.

5 Accuracy of Prediction Models

We conducted a preliminary study to assess the prediction models in *AutoPyDep*. Our evaluation set out to answer the following research question: *How accurately can the nature and timing of package releases be predicted based on historical release records?*

5.1 Predicting Release Categories

We evaluated the performance of our model in predicting release categories (Bug Fixes, New Features, Performance Improvements, and Security) using precision, recall, and F1-score metrics with 10-fold cross-validation. The model achieved F1-scores of 0.94 for Bug Fixes, 0.95 for New Features, 0.89 for Performance Improvements, and 0.92 for Security, as shown in Table 1. The micro, macro, and

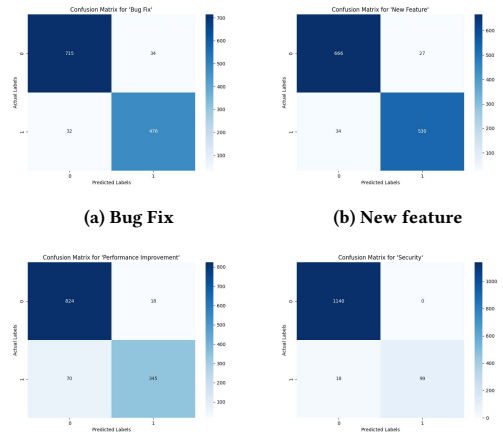
[4] https://pypi.org/project/customtkinter/0.3/

Table 1: Performance of Mod2(Community+eigenvalue) in release notes dataset

Category	Precision	Recall	F1-Score	Support
Bug Fix	0.93	0.94	0.94	508
New Feature	0.95	0.94	0.95	564
Performance Improvement	0.95	0.83	0.89	415
Security	1.00	0.85	0.92	117
micro avg	0.95	0.90	0.93	1604
macro avg	0.96	0.89	0.92	1604
weighted avg	0.95	0.90	0.92	1604
samples avg	0.94	0.93	0.93	1604

weighted F1 averages indicate robust predictive capability across all categories.

Confusion matrices in Figure 2 highlight category-specific performance. Stratified cross-validation results (Figure 3) show consistent F1-score distributions across folds, with a median F1 score of 0.8, demonstrating the model's generalizability and reliability in multi-label classification.



(a) Bug Fix **(b) New feature**

(c) Performance improvement **(d) Security**

Figure 2: Confusion Matrices for Four Labels

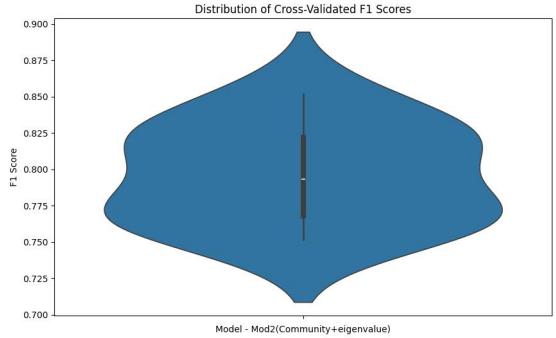


Figure 3: Cross-validation result distribution

5.2 Predicting Release Dates

To predict release dates, we used Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) metrics. Predictions, calculated in a year-month format, achieved an MAE of 1.8 months and an RMSE of 2 months. These results demonstrate the model's accuracy in estimating release intervals, even with larger deviations.

Summary

Our category prediction model achieved F1-scores of 0.94 for Bug Fixes, 0.95 for New Features, 0.89 for Performance Improvements, and 0.92 for Security, with a median cross-validated F1 score of 0.8. For the date prediction model, we achieved an MAE of 1.8 months and an RMSE of 2 months.

6 Discussion

AutoPyDep is designed for maintainers and researchers seeking to analyze and manage Python dependencies using release-driven signals. It can aid in identifying critical libraries, forecasting release impacts, and understanding ecosystem structure. While our current graph uses 23 libraries, the underlying architecture (embedding + graph-based analytics) is scalable—embeddings and graph computations can be parallelized, and community detection methods like Louvain remain efficient on graphs with thousands of nodes.

7 Future Work

For future work, we aim to conduct a user study to assess the usability and effectiveness of *AutoPyDep* for Python dependency management tasks, focusing on refining its features based on developer feedback. This study will also help ensure the tool aligns with practical needs to support dependency management across diverse development environments.

8 Threats to Validity

Our findings may face limitations in *external validity*, as the dataset comprises 23 Python libraries, which may not fully represent the broader Python ecosystem. Efforts were made to collect diverse data sources to mitigate this. We also only focus on Python, and future work is needed to evaluate the effectiveness of our approach for dependency management in other programming languages. *Internal validity* concerns include potential inconsistencies in release notes, addressed through thorough preprocessing and validation.

9 Conclusion

In this paper, we introduced *AutoPyDep*, a recommendation tool designed to streamline Python library dependency management. Key contributions include a novel dataset of Python release notes, predictive models for library update categories and release dates, and a recommendation system leveraging graph-based insights. The tool demonstrated strong performance in prediction tasks, providing actionable recommendations to assist developers in maintaining up-to-date and reliable software. Future work will focus on enhancing usability and expanding *AutoPyDep* to support additional programming languages and development environments.

References

[1] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 547–551. https://doi.org/10.1109/SANER48275.2020.9054837

[2] Ol' Al. 2024. How to avoid Dependency Hell? *DailyBot* (2024). https://www.dailybot.com/insights/how-to-avoid-dependency-hell-43549.

[3] John Bintz. 2019. Dependency hell: a complete guide. *Tidelift Blog* (2019). https://blog.tidelift.com/dependency-hell.

[4] Dibyendu Brinto Bose and Chris Brown. 2024. An Empirical Study on Current Practices and Challenges of Core AR/VR Developers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 233–238.

[5] Chris Brown and Chris Parnin. 2020. Comparing different developer behavior recommendation styles. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 78–85.

[6] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.

[7] Niels Cautaerts. 2024. Python dependency management is a dumpster fire. *nielscautaerts.xyz* (2024). https://nielscautaerts.xyz/python-dependency-management-is-a-dumpster-fire.html.

[8] Stephanie Dick and Daniel Volmar. 2018. DLL Hell: Software Dependencies, Failure, and the Maintenance of Microsoft Windows. *IEEE Annals of the History of Computing* 40, 4 (2018), 28–51. https://doi.org/10.1109/MAHC.2018.2877913

[9] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 463–474. https://doi.org/10.1145/3395363.3397388

[10] Marko Gasparic and Andrea Janes. 2016. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software* 113 (2016), 101–113.

[11] Wei Li, Justin Matejka, Tovi Grossman, Joseph A Konstan, and George Fitzmaurice. 2011. Design and evaluation of a command recommendation system for software applications. *ACM Transactions on Computer-Human Interaction (TOCHI)* 18, 2 (2011), 1–35.

[12] Lucy Ellen Lwakatare, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. 2019. A taxonomy of software engineering challenges for machine learning systems: An empirical investigation. In *Agile Processes in Software Engineering and Extreme Programming: 20th International Conference, XP 2019, Montréal, QC, Canada, May 21–25, 2019, Proceedings 20*. Springer International Publishing, 227–243.

[13] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. 2011. An empirical study of build maintenance effort. In *Proceedings of the 33rd international conference on software engineering*. 141–150.

[14] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 331–336.

[15] Francis Palma, Hadi Farzin, Yann-Gaël Guéhéneuc, and Naouel Moha. 2012. Recommendation system for design patterns in software development: An dpr overview. In *2012 third international workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 1–5.

[16] Lohith Paripati, Venudhar Rao Hajari, Narendra Narukulla, Nitin Prasad, Jigar Shah, and Akshay Agarwal. 2024. AI Algorithms for Personalization: Recommender Systems, Predictive Analytics, and Beyond. *Darpan International Research Analysis* 12, 2 (2024), 51–63.

[17] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. 2005. Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 167–176.

[18] Yudai Tanabe, Tomoyuki Aotani, and Hidehiko Masuhara. 2018. A Context-Oriented Programming Approach to Dependency Hell. In *Proceedings of the 10th ACM International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition* (Amsterdam, Netherlands) *(COP '18)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3242921.3242923

[19] Ferdian Thung. 2016. API recommendation system for software development. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 896–899.

[20] Z. Wan, X. Xia, and A. E. Hassan. 2019. What is Discussed about Blockchain? A Case Study on the Use of Balanced LDA and the Reference Architecture of a Domain to Capture Online Discussions about Blockchain platforms across the Stack Exchange Communities. *IEEE Transactions on Software Engineering* (2019), 1–1.