A GNN Based Recommendation System for Open-Source Contribution

Hamed Nasrolahi
George Mason University
Fairfax, VA 22030
hnasrola@gmu.edu

December 10, 2025

1 Introduction

Open Source Software (OSS) has evolved into a vast ecosystem where developers across the globe collaborate without formal hierarchies to collectively advance shared projects. Currently, GitHub stands as a pivotal platform for this development. In this environment, many developers, from junior to senior levels, contribute to open projects alongside their primary employment duties. While some contribute for leisure, many others do it in an attempt to signal their technical abilities for possible future employment. Anecdotal evidence suggests that such public demonstrations as the GitHub profile of job candidates has become a crucial component of their professional portfolio. As prior studies have also shown, career advancement in software development depends significantly on the nature and extent of these OSS contributions (Hann et al., 2013).

However, due to the rapid expansion of the OSS ecosystem, developers, especially those in their early career as programmers, often face difficulties in navigating the vast variety of available projects. It is observed that developers might become confused about which projects best suit their specific domain knowledge and experience. Currently, there is a lack of existing tools that effectively recommend projects to developers who seek to make contributions, often resulting in them spending a significant amount of time searching for a proper fit.

In this work, we propose a personalized and context-aware recommendation system to address this issue. We believe that a Graph Neural Network (GNN) is a specially suitable method for this task, as it leverages structural relationships such as collaboration frequencies, who-contributed-to-which-repo and has the potential to embed additional metadata of both developers and projects as features to precisely map skills to project requirements. By building a graph of users and repositories and learning node representations to predict edges, developers can receive personalized suggestions about potential repositories that better align with their background. Thus, the proposed approach contributes toward a more efficient navigation of the OSS ecosystem, potentially reducing search costs and supporting more informed participation in contribution activities.

2 Problem Definition

In the current software industry, developer contributions to Open Source Software (OSS) are visible to a broad audience. Hiring managers can observe indications of developer skills, both interpersonal and technical. This facilitates the emergence of some online competition for job hunting. In a study by Daniel et al. (2025) on 269 OSS developer, they showed that a developer making their technical skills visible on digital platforms is associated with job promotions. Consequently, developers are highly motivated to explore opportunities to contribute to projects that align with their career goals.

However, identifying these opportunities is not a trivial task. While GitHub provides search functionalities and star-based rankings, such mechanisms are not personalized. Since they often lack contextual information regarding the specific expertise of the user, they do not necessarily correspond to a developer's skills or past contribution history. This creates a gap between the user's intent to contribute and the discovery of suitable repositories.

Our methodology to bridge this gap is inspired by a recent Stanford machine learning project by Plonski and Cadicamo (2025), where Graph Neural Networks (GNNs) were used to recommend relevant academic papers for startup ideas expressed in natural language. We recognized that the tools and libraries used in their project are also applicable to this project. Therefore, to address the lack of personalization in current tools, we model the GitHub ecosystem as a graph where users and repositories are represented as nodes, and interactions (specifically code commits) form the edges.

In general, we utilize a bipartite graph structure. This means the graph consists of two disjoint sets of nodes: User nodes (U) and Repository nodes (R). An edge exists between a user U and a repository R if the user has committed to that repo (Figure 1). By leveraging the power of GNNs, we aim to predict which repositories a user is most likely to contribute to next. This prediction incorporates not only their individual collaboration history but also the history of similar users within the graph structure.

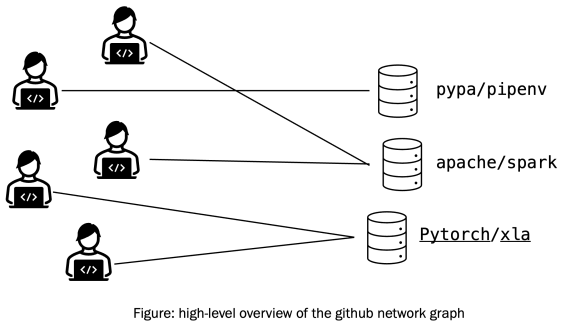


Figure 1: High-level overview of the github network graph.

3 Methodology

Our methodology consists of several key stages: data collection and preprocessing, graph construction, feature engineering, model training, and evaluation.

3.1 Data Collection and Preprocessing

Given the vast scale of GitHub, we limited our scope specifically to the Python ecosystem, as it represents one of the most active communities for open-source collaboration. We utilized a list of approximately 600,000 Python packages extracted from PyPI.org as our target repository set.

For interaction data, we accessed the GhArchive[1] dataset, which hosts hourly archives of all public GitHub events. We processed the log files for the first three months (Q1) of 2022. From the compressed JSON files, we extracted only the PushEvent activities associated with the repositories in our target list. For each event, we extracted the username, repository name, and timestamp. We then aggregate these events to construct user-repository interaction pairs.

Because raw GitHub data contains significant noise, to ensure our graph reflects genuine human collaboration, we applied several filtering rules including: Bot Removal: where filtered out actors identified as automated bots. Commit Validation: where we only counted PushEvents where the distinct size was 1 or greater, ensuring that the event represented an actual code contribution rather than an empty push or merge artifact. Active Days: instead of raw commit counts, which can be skewed by high-frequency bursts, we defined our edge weight based on "Active Days." If a user commits ten times in one day, it counts as 1 active day; if they commit once across five different days, it signals sustained involvement and counts as 5. Self-Contribution Removal: we also removed edges where a user committed to a repository they own, as our goal is to recommend new external projects to developers, not their own work.

3.2 Graph Construction

Using the processed data, we constructed an undirected bipartite graph containing two node types: Users (U) and Repositories (R). The edges represented the collaboration, weighted by the number of active days.

Upon initial inspection, the network structure was found to be uniquely sparse. While there was a large core of interconnected nodes, there were numerous small, isolated components that did not bear sufficient information for learning. To address this, we refined the network by removing components with fewer than 10 nodes.

3.3 Feature Engineering

Node features are critical for the GNN to learn meaningful embeddings. For all the remaining unique nodes in the refined graph, we queried GitHub's GraphQL API to extract rich metadata for node embeddings: User Features: biography, account creation date (used as a proxy for experience), number of followers, and number of owned repositories.

Repository Features: description, star count, fork count, watch count, and primary programming language.

To prepare this data for the model, we applied distinct preprocessing steps based on the data type:

Numerical Features: To handle the highly skewed distributions typical of social platform metrics (e.g., follower counts or stars), we applied a logarithmic transformation to all numerical data points.

[1] https://www.gharchive.org/

Textual Features: We encoded textual data (such as user biographies and repository descriptions) using the sentence-transformers model all-MiniLM-L6-v2[2]. This model produces fixed-length embeddings (384 dimensions) regardless of the input text length, capturing semantic meaning effectively.

Categorical Features: For the primary programming language, we utilized one-hot encoding across 9 categories: Python, C, C++, Rust, Cython, JavaScript, TypeScript, Jupyter Notebook, and Unidentified.

Based on these processing steps, the final input vectors for each user and repo node became a 387 and 396-dimensional feature vector respectively. The final graph contains 8,772 nodes (5,479 users and 3,293 repositories) connected by 9,954 edges.

4 Experiments

4.1 Model Architecture

Our baseline approach consists of two main components: an encoder that generates node embeddings through message passing, and a decoder that predicts the likelihood of user-repository interactions (Figure 2).

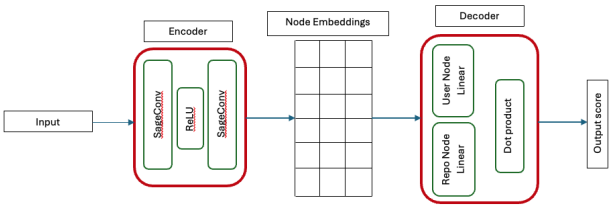


Figure 2: Baseline GNN Model

Encoder We employ a two-layer GraphSAGE (Graph Sample and Aggregation) architecture as our encoder. GraphSAGE is particularly suitable for bipartite graphs because it aggregates information from neighboring nodes regardless of their type. Our encoder processes the heterogeneous graph structure where users and repositories have different feature dimensions:

Input: x (node features), edge_index (graph structure)
Layer 1: h1 = SAGEConv(x, edge_index) h1 = ReLU(h1)
Layer 2: h2 = SAGEConv(h1, edge_index)
Output: h2 (node embeddings)

Decoder The decoder component predicts interaction probabilities between users and repositories based on their learned embeddings. We implement a simple yet effective dot-product-based decoder with learned transformations:

[2]https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

Input: z_user (user embeddings), z_repo (repo embeddings)
User transformation: u = Linear(z_user)
Repository transformation: r = Linear(z_repo)
Edge score: score = sum(u * r)
Output: score (prediction before sigmoid)

A critical design consideration in our bipartite graph is how the model processes two different node types with different feature dimensions. We address this through PyTorch Geometric's to_hetero transformation, which automatically creates separate learnable weight matrices for each node type. This allows the model to learn type-specific transformations: one set of parameters processes user features, and a completely independent set processes repository features, even though they're part of the same graph structure. For our bipartite graph, this means that user nodes receive their own set of weight matrices that transform 387-dimensional input features, and repository nodes receive a separate set of weight matrices that transform 396-dimensional input features. Both transformations produce outputs in the same 64-dimensional embedding space. This happens at each layer of the network. In Layer 1, user features (387 dimensions) pass through user-specific weights to produce 64-dimensional hidden representations, while repository features (396 dimensions) pass through repository-specific weights to produce their own 64-dimensional hidden representations. The same pattern repeats in Layer 2, where both node types have dedicated transformation parameters.

Edge Weight During graph construction, we computed edge weights based on number of active days. However, our baseline GraphSAGE architecture does not explicitly incorporate these edge weights during message passing. The rationale for this is that unweighted aggregation leads to a simpler architecture which is suitable for a baseline model. Adding weighted message passing introduces additional hyperparameters and complexity that may not be necessary for initial validation of the approach.

4.2 Model Training

Data Splitting A critical challenge in our sparse bipartite graph is ensuring all nodes remain connected during training. The mean user degree is only 1.82, meaning most users contributed to just one or two repositories. Similarly, 58% of repositories have exactly 2 contributors. This creates a situation where many nodes have very few connections, making them vulnerable to isolation if we randomly remove edges for validation and testing. We develop a custom splitting strategy to address this issue:

Calculate the degree of each node (both users and repositories). Categorize edges as "critical" if either endpoint has degree 1, meaning removing that edge would isolate a node. 1. Categorize remaining edges as "safe" for splitting. 2. Keep all critical edges in the training set to guarantee connectivity. 3. Randomly split only the safe edges into validation (10%) and test (10%) sets. 4. The training set consists of all critical edges plus the remaining 80% of safe edges.

Loss Functions We use two different loss functions to compare their effectiveness for the link prediction task of our model:

Binary Cross-Entropy (BCE): Binary Cross-Entropy (BCE) is a widely used loss function

in classification tasks, particularly when the goal is to predict binary outcomes (e.g., whether a user will interact with an item or not). For each training batch, we sample negative edges equal to the number of positive edges and compute:

$$\mathcal{L}\text{BCE} = -\sum \left[y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i) \right] \tag{1}$$

Bayesian Personalized Ranking (BPR): Bayesian Personalized Ranking loss is a loss function commonly used in recommendation systems, especially when the goal is to optimize the ranking of items for a user. The loss function is:

$$\mathcal{L}_{\text{BPR}} = -\sum_{(u,i,j)} \log \sigma \left(\hat{r}_{ui} - \hat{r}_{uj} \right) \tag{2}$$

Where:

- u is a user.
- i is a positive item (an item the user has interacted with).
- j is a negative item (an item the user has **not** interacted with).
- $\hat{r}_{ui}$ is the predicted score for user u and item i.
- $\hat{r}_{uj}$ is the predicted score for user u and item j.
- $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function, mapping scores to probabilities.

Evaluation Metrics We evaluate model performance using multiple metrics:

AUC-ROC: Measures the model's ability to distinguish positive edges from negative edges across all threshold values.
Precision@K: Among the top K recommended repositories for each user, what fraction are actually relevant?

$$\text{Precision@K} = \frac{\text{\# of relevant repos in top K}}{K}$$

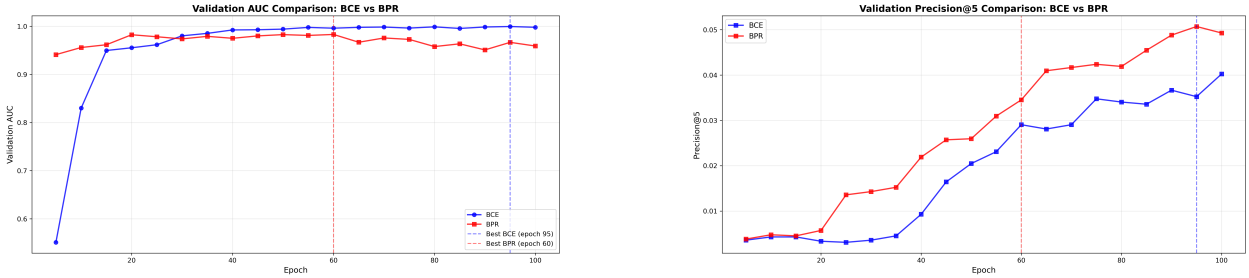
Recall@K: Among all relevant repositories for each user, what fraction appear in the top K recommendations?

$$\text{Recall@K} = \frac{\text{\# of relevant repos in top K}}{\text{total \# of relevant repos}}$$

We report metrics at K=5 and K=20 to evaluate performance at different recommendation list sizes.

4.3 Results

We trained our baseline model using the two different loss functions. Figure 3 shows the result plots for over 100 epochs. Both loss functions demonstrate smooth convergence throughout training process. By epoch 100, BPR achieves a lower training loss (approximately 0.04) compared to BCE (approximately 0.15). Figure 3a presents the validation AUC curves for both loss functions throughout training. The strange observation here is that both models jump to near-perfect AUC values while showing moderate Recall@5. This discrepancy reflects the fundamental difference between these metrics in sparse recommendation scenarios, and the explanation is that AUC measures the model's ability to distinguish true contributions from random non-contributions in pairwise comparisons. With graph density of 0.055%, the vast majority of user-repo pairs are true negatives that are easily identifiable, resulting in high AUC. In contrast, Recall@K evaluates ranking quality across ALL 3,293 repositories. For a user who contributed to just 1-2 repositories (our mean user degree is 1.82), finding these specific repositories in the top-5 requires beating 3,288 competing recommendations. Many of these competitors are popular repositories with similar features, making the task substantially harder than binary classification.



(a) Validation AUC

(b) Validation Precision@5

(c) Validation Recall@5

(d) Training Loss

Figure 3: Comparison of BCE and BPR loss functions across key metrics over 100 epochs. Overall, BCE shows superior stability and performance in validation metrics.

Figure 4 presents the final test set results for both models. First, both models achieve near-perfect AUC scores. While these numbers appear impressive, they unfortunately reflect a limitation of our evaluation setup rather than exceptional model performance. Again, the AUC evaluation uses randomly sampled negative edges, and with our graph density of only 0.055%, these negative samples are almost always obviously unrelated to the user. The model learns to distinguish true contributions from random non-contributions, a task that the sparse graph structure makes relatively straightforward. This means our AUC scores are somewhat inflated and do not fully capture the

model's ability to make fine-grained distinctions between similar repositories. The Recall@K metrics at different K values reveal how performance changes as we expand the recommendation list. At K=5, both models show modest recall (12.65% for BCE, 10.61% for BPR), meaning only about 1 in 8-10 true repositories appear in the top 5 recommendations. This reflects the difficulty of finding 1-2 true contributions among 3,293 candidates when many similar repositories compete for top positions. At K=20, recall nearly triples to 35.49% for BCE and 33.39% for BPR, indicating that expanding from 5 to 20 recommendations substantially increases the likelihood of including relevant repositories. The improvement from K=5 to K=20 suggests that while the model may not always rank the true repository at the very top, it frequently places it within the top 20; a reasonable performance given the sparse data and large candidate pool.

Model	AUC	P@5	R@5	P@20	R@20	P@100	R@100
Baseline BCE	0.9963	0.0253	0.1265	0.0177	0.3549	0.0087	0.8669
Baseline BPR	0.9741	0.0212	0.1061	0.0167	0.3339	0.0087	0.8705

Figure 4: Test set results

5 Future Work

This project was an experiment to design a tool that addresses a real world problem. However, the actual structure of network proved that the dynamics of open-source contribution world is much more complex than we thought which made the results of this endeavor disappointing. Nevertheless, we can think of are several promising directions that could improve and reform this project's objectives: our current approach uses uniform random sampling to select negative edges during training, producing "easy negatives" that are often completely unrelated to the user. Hard negative sampling would select negative examples that are similar to positive examples but not actually contributed to. For instance, for a machine learning researcher who contributed to PyTorch, we might sample negative examples from other ML frameworks like TensorFlow or rather than random repositories. This would force the model to learn more discriminative features that can distinguish between similar but non-interacted items. Implementation could be based on shared programming languages, similar description embeddings, or repositories popular among similar users. We expect this would reduce AUC (as negatives become harder) but improve Recall@K by teaching the model finer distinctions in top-ranked recommendations.

Our baseline model treats all edges equally during message passing, ignoring the contribution intensity information encoded in edge weights. Users who made a single commit influence their embedding identically to core maintainers with hundreds of commits. Future work could incorporate edge weights through Graph Attention Networks (GAT) that learn attention weights dynamically, or by modifying GraphSAGE aggregation to use weighted sums where messages from high-weight edges receive more influence. These modifications would enable the model to recognize that a user's primary projects should influence their embedding more strongly than casual contributions, potentially improving recommendation quality for users with varied contribution patterns.

Finally, our current dataset contains approximately 10,000 edges over a 3-month period. Expanding to longer time periods or broader repository coverage would provide more training signal and enable evaluation of how model performance scales with data size. We could extend our temporal window from 3 months to 6 or 12 months, potentially doubling or quadrupling our edge count. We could also expand beyond Python packages to include all GitHub repositories as larger datasets

would enable more confident evaluations.

Code Availability and Acknowledgments: All source code developed for this project is available at GitHub. The code implementation was primarily developed by the author with assistance from AI-powered coding tools including GitHub Copilot and Claude AI assistant integrated within Visual Studio Code. These tools were used for code completion, debugging, and implementation suggestions.

References

Daniel, Sherae L, Renzhi (Fred) Zhao, and Likoebe M. Maruping (2025). "Digital Skill Visibility and Job Promotion of Open Source Software Developers". *Journal of Management Information Systems* 42.3, pp. 926–950.

Hann, Il-Horn, Jeffrey A. Roberts, and Sandra A. Slaughter (2013). "All Are Not Equal: An Examination of the Economic Returns to Different Forms of Participation in Open Source Software Communities". *Information Systems Research* 24.3, pp. 520–538.

Plonski, Charlie and Nate Cadicamo (2025). *Bridging Academia and Industry: Using Graph Neural Networks to Recommend Relevant Research Papers for Startup Ideas*. URL: `https://medium.com/stanford-cs224w/bridging-academia-and-industry-using-graph-neural-networks-to-recommend-relevant-research-papers-9e9a2434f376/`.