Code Recommendation for Open Source Software Developers

Yiqiao Jin
Georgia Institute of Technology
Atlanta, GA, USA
yjin328@gatech.edu

Yunsheng Bai
University of California, Los Angeles
Los Angeles, CA, USA
yba@cs.ucla.edu

Yanqiao Zhu
University of California, Los Angeles
Los Angeles, CA, USA
yzhu@cs.ucla.edu

Yizhou Sun
University of California, Los Angeles
Los Angeles, CA, USA
yzsun@cs.ucla.edu

Wei Wang
University of California, Los Angeles
Los Angeles, CA, USA
weiwang@cs.ucla.edu

ABSTRACT

Open Source Software (OSS) is forming the spines of technology infrastructures, attracting millions of talents to contribute. Notably, it is challenging and critical to consider both the developers' interests and the semantic features of the project code to recommend appropriate development tasks to OSS developers. In this paper, we formulate the novel problem of code recommendation, whose purpose is to predict the future contribution behaviors of developers given their interaction history, the semantic features of source code, and the hierarchical file structures of projects. We introduce CODER, a novel graph-based CODE Recommendation framework for open source software developers, which accounts for the complex interactions among multiple parties within the system. CODER jointly models microscopic user-code interactions and macroscopic user-project interactions via a heterogeneous graph and further bridges the two levels of information through aggregation on file-structure graphs that reflect the project hierarchy. Moreover, to overcome the lack of reliable benchmarks, we construct three large-scale datasets to facilitate future research in this direction. Extensive experiments show that our CODER framework achieves superior performance under various experimental settings, including intra-project, cross-project, and cold-start recommendation.

CCS CONCEPTS

• **Information systems** → **Collaborative filtering**; *Web and social media search*; **Social recommendation**; *Personalization.*

KEYWORDS

Code recommendation; recommender system; open source software development; multimodal recommendation; graph neural networks

ACM Reference Format:
Yiqiao Jin, Yunsheng Bai, Yanqiao Zhu, Yizhou Sun, and Wei Wang. 2023. Code Recommendation for Open Source Software Developers. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30-May 4, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3543507.3583503

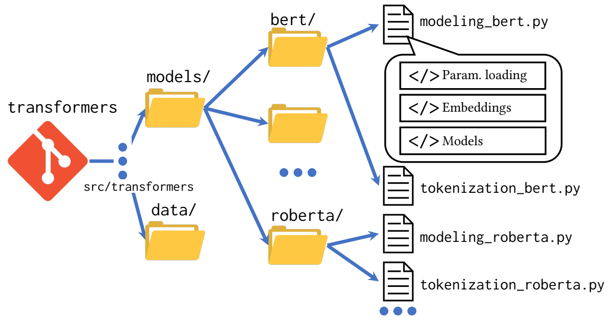

Figure 1: An example of the `transformers` repository. OSS projects under similar topics usually adopt similar naming conventions and file structures, which can be seen as knowledge transferable across projects.

1 INTRODUCTION

Open Source Software (OSS) is becoming increasingly popular in software engineering [23, 50]. As contribution to OSS projects is highly democratized [70], these projects attract millions of developers with diverse expertise and efficiently crowd-source the project development to a larger community of developers beyond the project's major personnel [23, 35]. For instance, GitHub, one of the most successful platforms for developing and hosting OSS projects, has over 83 million users and 200 million repositories [13].

Community support and teamwork are major driving forces behind open source projects [35]. OSS projects are usually developed in a collaborative manner [2], whereas collaboration in OSS is especially challenging. OSS projects are of large scales and usually contain numerous project files written in diverse programming languages [4]. According to statistics, the most popular 500 GitHub projects contain an average of 2,582 project files, 573 directories, and 360 contributors. Meanwhile, there are more than 300 programming languages on GitHub, 67 of which are actively being used [11, 12]. For **project maintainers**, it is both difficult and time-consuming to find competent contributors within a potentially large candidate pool. For **OSS developers**, recommending personalized development tasks according to their project experience and expertise can significantly boost their motivation and reduce their cognitive loads of manually checking the project files. As contribution in OSS is voluntary, developers that fail to find meaningful tasks are likely to quit the project development [48]. Therefore, an efficient system for automatically matching source code with potential contributors is being called for by both the project core team and the potential contributors to reduce their burden.

To solve the above issues, in this paper, we for the first time introduce the novel problem of code recommendation for OSS developers. As shown in Fig. 2, this task recommends code in the form of project files to potentially suitable contributors. It is noteworthy that code recommendation has several unique challenges such that traditional recommender models are not directly applicable.

Firstly, OSS projects contain multimodal interactions among users, projects, and code files. For example, OSS development contains user-code interactions, such as commits that depict microscopic behaviors of users, and user-project interactions, such as forks and stars that exhibit users' macroscopic preferences and interests on projects. Also, the contribution relationships are often extremely sparse, due to the significant efforts required to make a single contribution to OSS projects. Therefore, directly modeling the contribution behavior as in traditional collaborative filtering approaches will inevitably lead to inaccurate user/item representations and suboptimal performances.

Secondly, in the software engineering domain, code files in a project are often organized in a hierarchical structure [69]. Fig. 1 shows an example of the famous `huggingface/transformers` repository [56]. The `src` directory usually contains the major source code for a project. The `data` and `models` subdirectories usually include functions for data generation and model implementations, respectively. Such a structural organization of the OSS project reveals semantic relations among code snippets, which are helpful for developers to transfer existing code from other projects to their development. Traditional methods usually ignore such item-wise hierarchical relationships and, as a result, are incapable of connecting rich semantic features in code files with their project-level structures, which is required for accurate code recommendation.

Thirdly, most existing benchmarks involving recommendation for softwares only consider limited user-item behaviors [5, 21], are of small scales [39, 40], or contain only certain languages such as Python [20, 37, 51] or Java [5, 21, 40], which renders the evaluation of different recommendation models difficult or not realistic.

To overcome the above challenges, we propose **CODER**, a CODE Recommendation framework for open source software developers that matches project files with potential contributors. As shown in Fig. 2, CODER treats users, code files, and projects as nodes and jointly models the microscopic user-code interactions and macroscopic user-project interactions in a heterogeneous graph. Furthermore, CODER bridges these two levels of information through message aggregation on the file structure graphs that reflect the hierarchical relationships among graph nodes. Additionally, since there is a lack of benchmark datasets for the code recommendation task, we build three large-scale datasets from open software development websites. These datasets cover diverse subtopics in computer science and contain up to 2 million fine-grained user-file interactions. Overall, our contributions are summarized as follows:

- We for the first time introduce the problem of code recommendation, whose purpose is to recommend appropriate development tasks to developers, given the interaction history of developers, the semantic features of source code, and hierarchical structures of projects.
- We propose CODER, an end-to-end framework that jointly models structural and semantic features of source code as well as multiple types of user behaviors for improving the matching task.
- We construct three large-scale multi-modal datasets for code recommendation that cover different topics in computer science to facilitate research on code recommendation.
- We conduct extensive experiments on massive datasets to demonstrate the effectiveness of the proposed CODER framework and its design choices.

2 PRELIMINARIES

2.1 GitHub

GitHub is a code hosting platform for version control and collaboration based on `git`. Users can create `repositories`, namely, digital directories, to store source code for their projects. Users can make changes to the source code in the form of `commits`, which are snapshots of an OSS project, that capture the project state. A GitHub `commit` mainly reflects two aspects: 1) The commit author is highly interested in the project; 2) the user has the expertise to contribute. In this work, we study the factors that influence users' contributing behaviors. We use git `commits` as positive interactions.

2.2 Problem Formulation

Before delving into our proposed CODER framework, we first formalize our code recommendation task. We use the terms "repository" and "project" interchangeably to refer to an open source project. We define $\mathcal{U}$, $\mathcal{V}$, $\mathcal{R}$ as the set of users, files, and repositories, respectively. Each repository $r_k \in \mathcal{R}$ contains a subset of files $\mathcal{V}_k \subsetneq \mathcal{V}$. Both macroscopic project-level interactions and microscopic file-level interactions are present in OSS development.

File-level behaviors. We define $\mathbf{Y} \in \{0,1\}^{|\mathcal{U}|\times|\mathcal{V}|}$ as the interaction matrix between $\mathcal{U}$ and $\mathcal{V}$ for the file-level contribution behavior, where each entry is denoted by y_{ij}. $y_{ij} = 1$ indicates that u_i has contributed to v_j, and $y_{ij} = 0$, otherwise.

Project-level behaviors. Interactions at the project level are more diverse. For example, the popular code hosting platform GitHub allows users to *star* (publicly bookmark) interesting repositories and *watch* (subscribe to) repositories for updates. We thus define $\mathcal{T}$ as the set of user-project behaviors. Similar to $\mathbf{Y}$, we define $\mathbf{S}_t \in \{0,1\}^{|\mathcal{U}|\times|\mathcal{R}|}$ as the project-level interaction matrix for behavior of type t. Our goal is to predict the future file-level contribution behaviors of users based on their previous interactions. Formally, given the training data $\mathbf{Y}^{\mathrm{TR}}$, we try to predict the interactions in the test set $y_{ij} \in \mathbf{Y}^{\mathrm{TS}} = \mathbf{Y} \backslash \mathbf{Y}^{\mathrm{TR}}$.

3 METHODOLOGY

As shown in Fig. 2, we design CODER, a two-stage graph-based recommendation framework. CODER considers $u_i \in \mathcal{U}, v_j \in \mathcal{V}, r_k \in \mathcal{R}$ as graph nodes, and models the user-item interactions and the item-item relations as edges. We use two sets of graphs to characterize the heterogeneous information in code recommendation. One is the user-item interaction graphs that encompass the collaborative signals. The other is the file-structure graphs that reveal file-file and

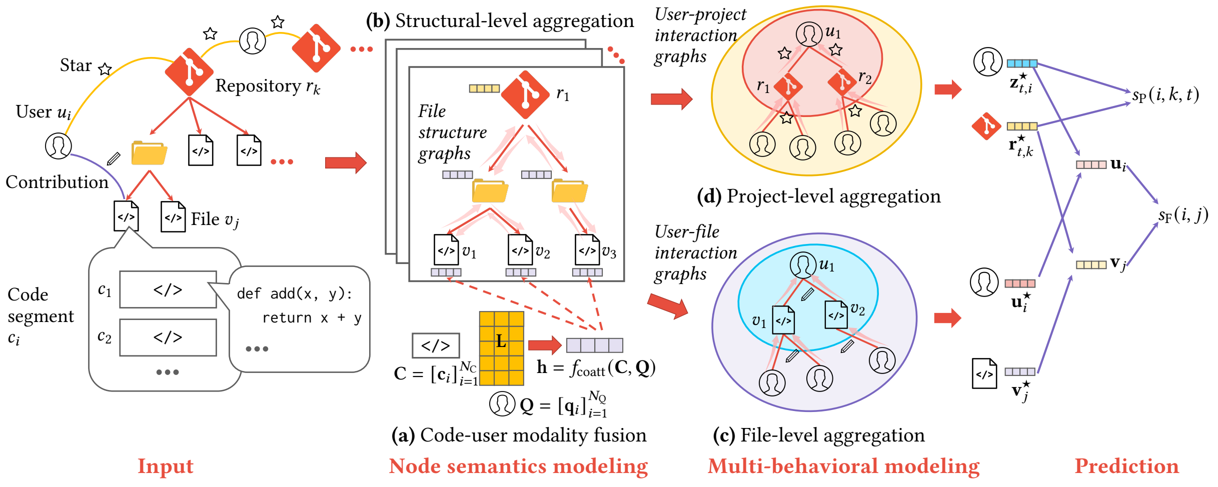
Star
User u_i
Repository r_k
Contribution
File v_j
Code segment c_i
c_1
c_2
def add(x, y):
return x + y
(b) Structural-level aggregation
File structure graphs
r_1
v_1 v_2 v_3
$\mathbf{C} = [\mathbf{c}_i]_{i=1}^{N_C}$
L
$\mathbf{h} = f_{\text{coatt}}(\mathbf{C}, \mathbf{Q})$
$\mathbf{Q} = [\mathbf{q}_i]_{i=1}^{N_Q}$
(a) Code-user modality fusion
User-project interaction graphs
u_1 r_1 r_2
(d) Project-level aggregation
User-file interaction graphs
u_1 v_1 v_2
(c) File-level aggregation
$\mathbf{z}^{\star}_{t,i}$
$\mathbf{r}^{\star}_{t,k}$
$s_P(i,k,t)$
$\mathbf{u}_i$
$s_F(i,j)$
$\mathbf{v}_j$
$\mathbf{u}^{\star}_i$
$\mathbf{v}^{\star}_j$
Input
Node semantics modeling
Multi-behavioral modeling
Prediction

Figure 2: Our proposed CODER framework for code recommendation. CODER jointly considers project file structures, code semantics, and user behaviors. CODER models the microscopic file-level interactions and macroscopic project-level interactions through Multi-Behavioral Modeling, and bridges the micro/macro-scopic signals through Node Semantics Modeling.

file-project relationships from the project hierarchy perspective. The code recommendation problem is then formulated as a user-file link prediction task.

CODER contains two major components: 1) *Node Semantics Modeling*, which learns the fine-grained representations of project files by fusing code semantics with their historical contributors, and then aggregate project hierarchical information on the file structure graph to learn the file and repository representation; 2) *Multi-behavioral Modeling*, which jointly models the microscopic user-file interactions and macroscopic user-project interactions. Finally, CODER fuses the representations from multiple behaviors for prediction. This way, node semantics modeling bridges the coarse-grained and fine-grained interaction signals on the item side. Therefore, CODER efficiently characterizes intra-project and inter-project differences, eventually uncovering latent user and item features that explain the interactions $\mathbf{Y}$.

3.1 Node Semantics Modeling

Node semantics modeling aims to learn file and repository representation. The challenge is how to inherently combine the semantic features of each project file with its interacted users and the project hierarchy. To address this challenge, we first use a **code-user modality fusion** mechanism (Fig. 2a) to fuse the file content modality and the historical users at the code level. Then, we embed the fine-grained collaborative signals from user-file interactions into the file representations. Next, we employ **structural-level aggregation** (Fig. 2b), which explicitly models the project structures as hierarchical graphs to enrich the file/repository representation with structural information. This step produces representation for each file v_j and repository r_k, which serve as the input for user behavior modeling in Sec. 3.2.

3.1.1 Code-User Modality Fusion. A project file is characterized by diverse semantic features including multiple method declarations and invocations, which are useful for explaining why a contributor is interested in it. Inspired by the success of pretrained language models [24, 26, 31], we use pretrained CodeBERT [7], a bimodal language model for programming languages, to encode the rich semantic features of each file. CodeBERT is shown to generalize well to programming languages not seen in the pretraining stage, making it suitable for our setting where project files are written in diverse programming languages. Here, a straightforward way is to directly encode each file into a per-file latent representation. Such an encoding scheme has two issues. Firstly, a file may contain multiple classes and function declarations that are semantically distinct. Fig. 1 shows the file structure of the `huggingface/transformers` [56] repository as an example. The `modeling_bert.py` file contains not only various implementations of the BERT language model for different NLP tasks, but also utilities for parameter loading and word embeddings. These implementations are distributed among several code segments in the same project file, and file-level encoding can fail to encode such semantic correlations. Secondly, the property of a project file is influenced by the historical contributors' features. A user's contribution can be viewed as injecting her/his own attributes, including programming style and domain knowledge, into the interacted file. Such contribution behaviors make it more likely to be interacted again by users with similar levels of expertise than random contributors.

Therefore, we propose a **code-user modality fusion** strategy to embed both code semantics and user characteristics into the file representation. Specifically, for each file, we partition its source code into N_C code segments and encode each of them into a code-segment-level representation $\mathbf{c}_i$. This produces a feature map $\mathbf{C} = [\mathbf{c}_1, \mathbf{c}_2, \dots \mathbf{c}_{N_C}], \mathbf{C} \in \mathbb{R}^{N_C \times d}$, where d is the embedding size. Similarly, we sample N_Q historical users of the file and encode them into a feature map $\mathbf{Q} = [\mathbf{u}_1, \mathbf{u}_2, \dots \mathbf{u}_{N_Q}], \mathbf{Q} \in \mathbb{R}^{N_Q \times d}$. Please refer to Appendix B for details in encoding $\mathbf{C}$ and $\mathbf{Q}$. Inspired by the success of co-attention [32, 71], we transform the user attention space to code attention space by calculating a code-user affinity matrix $\mathbf{L} \in \mathbb{R}^{N_C \times N_Q}$:

$$\mathbf{L} = \tanh(\mathbf{C}\mathbf{W}_O\mathbf{Q}^\top), \quad (1)$$

where $\mathbf{W}_O \in \mathbb{R}^{d \times d}$ is a trainable weight matrix. Next, we compute the attention weight $\mathbf{a} \in \mathbb{R}^{N_C}$ of the code segments to select salient features from $\mathbf{C}$. We treat the affinity matrix as a feature and learn an attention map $\mathbf{H}$ with N_H representations:

$$\mathbf{H} = \tanh(\mathbf{W}_C\mathbf{C}^\top + \mathbf{W}_Q(\mathbf{L}\mathbf{Q})^\top), \quad (2)$$

$$\mathbf{a} = \mathrm{softmax}(\mathbf{w}_H^\top\mathbf{H}), \quad (3)$$

where $\mathbf{W}_C, \mathbf{W}_Q \in \mathbb{R}^{N_H \times d}, \mathbf{w}_H \in \mathbb{R}^{N_H}$ are the weight parameters. Finally, the file attention representation $\mathbf{h}$ is calculated as the weighted sum of the code feature map:

$$\mathbf{h} = \mathbf{a}^\top\mathbf{C}. \quad (4)$$

The file attention representation serves as a start point to further aggregate file structural features.

3.1.2 *Structural-Level Aggregation.* Projects are organized in a hierarchical way such that nodes located closer on the file structure graph are more closely related in terms of semantics and functionality. For example, in Fig. 1, both files under the `bert/` directory contain source code for the BERT [26] language model, and files under `roberta/` contains implementation for the RoBERTa [31] model. The file `modeling_bert.py` is therefore more closely related to `tokenization_bert.py` in functionality than to `tokenization_roberta.py`.

To exploit such structural clues, we model each repository as a hierarchical heterogeneous graph [69] G_S consisting of file, directory, and repository nodes. Each node is connected to its parent node through an edge, and nodes at the first level are directly connected to the virtual root node representing the project. To encode the features of directory nodes, we partition the directory names into meaningful words according to underscores and letter capitalization, then encoded the nodes by their TF-IDF features. Our encoding scheme is motivated by the insight that the use of standard directory names (e.g., `doc`, `test`, and `models`) is correlated with project popularity among certain groups of developers [2, 79]. Repository features are encoded by their project owners, creation timestamps, and their top-5 programming languages. The repository and directory representations are mapped to the same latent space as the file nodes. Then, we pass the representation $\mathbf{h}$ through multiple GNN layers to aggregate the feature of each node from its neighbors on G_S:

$$\widetilde{\mathbf{h}} = f_{\mathrm{GNN}}(\mathbf{h}, G_S), \quad (5)$$

where $\widetilde{\mathbf{h}}$ is the structure-enhanced node representation. The aggregation function $f_{\mathrm{GNN}}(\cdot)$ can be chosen from a wide range of GNN architectures, such as GCN [28], GraphSAGE [16], and GIN [61]. In practice, we employ a 3-layer Graph Attention Network (GAT) [49].

3.2 Multi-behavioral Modeling

Direct modeling of the sparse contribution behavior potentially leads to inaccurate user/item representations and aggravates the cold-start issue. Instead, we jointly model the microscopic user-file contribution in File-level Aggregation (Fig. 2c) and macroscopic user-project interactions in Project-level Aggregation (Fig. 2d) to learn user preferences and address the sparsity issue. Then, the representations learned from multi-level behaviors are combined to form the user and item representations for prediction.

3.2.1 *File-level Aggregation.* We model the project files and their contributors as an undirected user-file bipartite graph $\mathcal{G}_F$ consisting of users $u_i \in \mathcal{U}$, files $v_j \in \mathcal{V}$ and their interactions. The initial embedding matrix of users/items is denoted by $\mathbf{E}^{(0)}$:

$$\mathbf{E}^{(0)} = [\underbrace{\mathbf{u}_1^{(0)}, \cdots, \mathbf{u}_{|\mathcal{U}|}^{(0)}}_{\text{users embeddings}}, \underbrace{\mathbf{v}_1^{(0)}, \cdots, \mathbf{v}_{|\mathcal{V}|}^{(0)}}_{\text{item embeddings}}], \quad (6)$$

where $\mathbf{u}_i^{(0)}$ is the initial embedding for user u_i and $\mathbf{v}_j^{(0)}$ is the initial embedding for file v_j, equivalent to its structure-enhanced representation $\widetilde{\mathbf{h}}$ (Sec. 3.1.2). We adopt the simple weight sum aggregator in LightGCN [18] in the propagation rule:

$$\mathbf{u}_i^{(l)} = \sum_{v_j \in \mathcal{N}_i} \frac{1}{\sqrt{|\mathcal{N}_i|\,|\mathcal{N}_j|}} \mathbf{v}_j^{(l-1)}, \quad (7)$$

$$\mathbf{v}_j^{(l)} = \sum_{u_i \in \mathcal{N}_j} \frac{1}{\sqrt{|\mathcal{N}_j|\,|\mathcal{N}_i|}} \mathbf{u}_i^{(l-1)}, \quad (8)$$

where $\mathbf{u}_i^{(l)}$ and $\mathbf{v}_j^{(l)}$ are the embeddings for user u_i and file v_j at layer l, $\mathcal{N}_i$ and $\mathcal{N}_j$ indicate the neighbors of user u_i and file v_j, and $1/\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}$ is the symmetric normalization term set to the graph Laplacian norm to avoid the increase of GCN embedding sizes [18, 28]. In the matrix form, the propagation rule of file-level aggregation can be expressed as:

$$\mathbf{E}^{(l)} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{E}^{(l-1)}, \quad (9)$$

where $\mathbf{A} = \begin{pmatrix} \mathbf{0} & \mathbf{Y}^{\mathrm{TR}} \\ (\mathbf{Y}^{\mathrm{TR}})^\top & \mathbf{0} \end{pmatrix} \in \mathbb{R}^{(|\mathcal{U}|+|\mathcal{V}|)\times(|\mathcal{U}|+|\mathcal{V}|)}$ is the affinity matrix, and $\mathbf{D}$ is the diagonal degree matrix in which each entry $\mathbf{D}_{ii}$ indicates the number of non-zero entries on the i-th row of $\mathbf{A}$. By stacking multiple layers, each user/item node aggregates information from its higher-order neighbors. Propagation through L layers yields a set of representations $\{\mathbf{E}^{(l)}\}_{l=0}^{L}$. Each $\mathbf{E}^{(l)}$ emphasizes the messages from its l-hop neighbors. We apply mean-pooling over all $\mathbf{E}^{(l)}$ to derive the user and file representations $\mathbf{u}_i^\star$ and $\mathbf{v}_j^\star$ from different levels of user/item features:

$$\mathbf{u}_i^\star = \frac{1}{L+1}\sum_{l=0}^{L}\mathbf{u}_i^{(l)}, \quad (10)$$

$$\mathbf{v}_j^\star = \frac{1}{L+1}\sum_{l=0}^{L}\mathbf{v}_j^{(l)}. \quad (11)$$

3.2.2 *Project-Level Aggregation.* OSS development is characterized by both microscopic contribution behaviors and multiple types of macroscopic project-level behaviors. For example, developers usually find relevant projects and reuse their functions and explore ideas of possible features [20, 22]. In particular, GitHub users can *star* (bookmark) interesting repositories and discover projects under similar topics. This way, developers can adapt code implementation of these interesting projects into their own development later. Hence, project-level macroscopic interactions are conducive for extracting users' broad interests.

For each behavior t, we propagate the user and repository embeddings on its project-level interaction graph $\mathcal{G}_{\mathrm{P}}^t$:

$$\mathbf{Z}_t^l = \mathbf{D}_t^{-\frac{1}{2}} \mathbf{\Lambda}_t \mathbf{D}_t^{-\frac{1}{2}} \mathbf{Z}_t^{(l-1)}. \tag{12}$$

The initial embeddings $\mathbf{Z}^{(0)}$ is shared by all $t \in \mathcal{T}$ and is composed of the initial user representations identical to Eq. (6) and the repository embeddings from the structure-enhanced representation $\widetilde{\mathbf{h}}$ in Eq. (5):

$$\mathbf{Z}^{(0)} = [\underbrace{\mathbf{z}_1^{(0)}, \mathbf{z}_2^{(0)}, \ldots \mathbf{z}_{|\mathcal{U}|}^{(0)}}_{\text{user embeddings}}, \underbrace{\mathbf{r}_1^{(0)}, \mathbf{r}_2^{(0)}, \ldots \mathbf{r}_{|\mathcal{R}|}^{(0)}}_{\text{repository embeddings}}], \tag{13}$$

where $\mathbf{z}_i^{(0)} = \mathbf{u}_i^{(0)}$, $\mathbf{\Lambda}_t \in \mathbb{R}^{(|\mathcal{U}|+|\mathcal{R}|)\times(|\mathcal{U}|+|\mathcal{R}|)}$ is the affinity matrix for behavior t constructed similarly as $\mathbf{A}$ in Eq. (9). With representations $\{\mathbf{Z}_t^{(l)}\}_{l=0}^L$ obtained from multiple layers, we derive the combined user and repository representations for behavior t as

$$\mathbf{z}_{t,i}^{\star} = \frac{1}{L+1} \sum_{l=0}^{L} \mathbf{z}_{t,i}^{(l)}, \tag{14}$$

$$\mathbf{r}_{t,i}^{\star} = \frac{1}{L+1} \sum_{l=0}^{L} \mathbf{r}_{t,i}^{(l)}. \tag{15}$$

3.3 Prediction

For file-level prediction, we aggregate the macroscopic signals $\mathbf{z}_{t,i}^{\star}, \mathbf{r}_{t,i}^{\star}$ from each behavior t into $\mathbf{u}_i, \mathbf{v}_j$:

$$\mathbf{z}_i^{\star} = \mathrm{AGG}(\{z_t^{\star}, t \in \mathcal{T}\}), \qquad \mathbf{r}_k^{\star} = \mathrm{AGG}(\{r_t^{\star}, t \in \mathcal{T}\}), \tag{16}$$

$$\mathbf{u}_i = \mathrm{MLP}([\mathbf{u}_i^{\star} \parallel \mathbf{z}_i^{\star}]), \qquad \mathbf{v}_j = \mathrm{MLP}([\mathbf{v}_j^{\star} \parallel \mathbf{r}_{\phi(j)}^{\star}]), \tag{17}$$

where AGG$(\cdot)$ is an aggregation function and MLP$(\cdot)$ is a multilayer perceptron, $\phi(\cdot): \mathcal{V} \rightarrow \mathcal{R}$ maps the index of each project file to its repository, and $\parallel$ is the concatenation operator. On the user side, both macroscopic interests and micro-level interactions are injected into the user representations. On the item side, the semantics of each file is enriched by its interacted users and the repository structural information.

For computational efficiency, we employ inner product to calculate the user u_i's preference towards each file v_j:

$$s_{\mathrm{F}}(i,j) = \mathbf{u}_i^{\top} \mathbf{v}_j, \tag{18}$$

where s_{F} is the scoring function for the file-level behavior. Similarly, for each user-project pair, we derive a project-level score for each behavior t using the project-level scoring function s_{P}:

$$s_{\mathrm{P}}(i,k,t) = (\mathbf{z}_{t,i}^{\star})^{\top} \mathbf{r}_{t,k}^{\star}. \tag{19}$$

3.4 Optimization

We employ the Bayesian Personalized Ranking (BPR) [43] loss, which encourages the prediction of an observed user-item interaction to be greater than an unobserved one:

$$\mathcal{L}_{\mathrm{F}} = \sum_{(i,j^+,j^-)\in O} -\log(\mathrm{sigmoid}(s_{\mathrm{F}}(i,j^+) - s_{\mathrm{F}}(i,j^-))), \tag{20}$$

$$\mathcal{L}_{\mathrm{P}}^t = \sum_{(i,k^+,k^-)\in O} -\log(\mathrm{sigmoid}(s_{\mathrm{P}}(i,k^+,t) - s_{\mathrm{P}}(i,k^-,t))), \tag{21}$$

where $\mathcal{L}_{\mathrm{F}}$ is the file-level BPR loss, and $\mathcal{L}_{\mathrm{P}}^t$ is the project-level BPR loss for behavior t, O denotes the pairwise training data, j^+ indicates an observed interaction between user u_i, and item v_{j^+} and j^- indicates an unobserved one. As high-order neighboring relations within contributors are also useful for recommendations, we enforce users to have similar representations as their structural neighbors through structure-contrastive learning objective [30, 57]:

$$\mathcal{L}_{\mathrm{C}}^{\mathcal{U}} = \sum_{u_i \in \mathcal{U}} -\log \frac{\exp(\mathbf{u}_i^{(\eta)} \cdot \mathbf{u}_i^{(0)}/\tau)}{\sum_{u_j \in \mathcal{U}} \exp(\mathbf{u}_i^{(\eta)} \cdot \mathbf{u}_j^{(0)}/\tau)}. \tag{22}$$

Here, η is set to an even number so that each user node can aggregate signals from other user nodes and τ is a temperature hyper-parameter. Similarly, the contrastive loss is applied to each v_i:

$$\mathcal{L}_{\mathrm{C}}^{\mathcal{V}} = \sum_{v_i \in \mathcal{V}} -\log \frac{\exp(\mathbf{v}_i^{(l)} \cdot \mathrm{v}_i^{(0)}/\tau)}{\sum_{v_j \in \mathcal{V}} \exp(\mathbf{v}_i^{(l)} \cdot \mathbf{v}_j^{(0)}/\tau)}. \tag{23}$$

The overall optimization objective is

$$\mathcal{L} = \mathcal{L}_{\mathrm{F}} + \lambda_1 \sum_{t\in\mathcal{T}} \mathcal{L}_{\mathrm{P}}^t + \lambda_2(\mathcal{L}_{\mathrm{C}}^{\mathcal{U}} + \mathcal{L}_{\mathrm{C}}^{\mathcal{V}}) + \lambda_3 \|\Theta\|_2, \tag{24}$$

where Θ denotes all trainable model parameters and $\lambda_1, \lambda_2, \lambda_3$ are hyper-parameters.

3.5 Complexity Analysis

The node semantic modeling (Sec. 3.1) has time complexity of $O((N_{\mathrm{C}} + N_{\mathrm{Q}})|\mathcal{V}| + |\mathcal{E}|)$, where $\mathcal{E}$ is the set of edges in all file structure graphs. The user behavior modeling (Sec. 3.2) has time complexity of $O(|\mathbf{A}^+| + \sum_t |\mathbf{A}_t^+|)$, where $|\mathbf{A}^+|$ is the number of positive entries in $|\mathbf{A}|$ and likewise for $|\mathbf{\Lambda}_t^+|$. The overall time complexity is $O((N_{\mathrm{C}} + N_{\mathrm{Q}})|\mathcal{V}| + |\mathcal{E}| + |\mathbf{A}^+| + \sum_t |\mathbf{\Lambda}_t^+|)$. Although obtaining the initial code segment embeddings $\mathbf{C}$ implies large computational costs, our model only calculates $\mathbf{C}$ once and caches it to be used in each iteration. Empirically, the average inference time of MF, LightGCN, and CODER are 0.804 ms, 1.073 ms, and 1.138 ms per test example, respectively.

4 EXPERIMENTS

4.1 Experimental Settings

4.1.1 Datasets. We collected 3 datasets covering diverse topics in computer science including machine learning (ML), fullstack (FS), and database (DB), using the GitHub API [1] and the PyGithub [2]package. We retain repositories with ≥ 250 stars and ≥ 3 contributors to exclude repositories intended for private usages [2]. We include projects with contribution history of at least 3 months according to their commit history. To ensure that our model generalizes on a wide range of topics, popularity, and project scales, we first select 3 subsets of repositories using their GitHub topics [3], which are project labels created by the project owners. Then, we randomly sample 300 repositories from each subset considering their numbers of project files and stars. We use the Unix timestamp `1550000000` and `1602000000` to partition the datasets into training, validation, and test sets. This way, all interactions before the timestamp are used

[1]https://docs.github.com/en/rest
[2]https://github.com/PyGithub/PyGithub.git
[3]https://github.com/topics

Table 1: Summary of the datasets. The second column shows the number of files with observed interactions instead of all existing files in the projects.

Dataset	#Files	#Users	#Interactions	Density
ML	239,232	21,913	663,046	1.26×10^{-4}
DB	415,154	30,185	1,935,155	1.54×10^{-4}
FS	568,972	51,664	1,512,809	5.14×10^{-5}

as the training data. We retain the users with at least 1 interaction in both train and test set. More details about dataset construction are in the appendix.

4.1.2 Implementation Details. We implemented our CODER model in PyTorch [42] and PyG [8]. For all models, we set the embedding size to 32 and perform Xavier initialization [14] on the model parameters. We use Adam optimizer [27] with a batch size of 1024. For Node Semantic Modeling (Sec. 3.1), we conduct a grid search on $N_C \in \{4, 8, 12, 16\}$ and $N_Q \in \{2, 4, 8\}$, and choose to set $N_C = 8$ and $N_Q = 4$. The code encoder we use is the pre-trained CodeBERT [7] model with 6 layers, 12 attention heads, and 768-dimensional hidden states. For Multi-Behavioral Modeling (Sec. 3.2), we set the number of convolution layers $L = 4$ for both intra- and inter-level aggregation. For prediction and optimization, we search the hyper-parameter λ_3 in $\{10^{-4}, 10^{-3}, 10^{-2}\}$, and λ_1 in $\{10^{-2}, 10^{-1}, 1\}$. For the structure contrastive loss [30], we adopt the hyper-parameter setting from the original implementation and set $\lambda_2 = 10^{-6}, \eta = 2$ without further tuning. For the baseline models, the hyper-parameters are set to the optimal settings as reported in their original papers. For all models, we search the learning rate in $\{10^{-4}, 3 \times 10^{-4}, 10^{-3}, 3 \times 10^{-3}, 10^{-2}\}$.

4.1.3 Baselines. We compare CODER with 3 groups of methods:

- **G1**: a factorization-based method MF [43],
- **G2**: neural-network-based methods including MLP [47] and Neu-MF [19],
- **G3**: graph-based methods that model user-item interactions as graphs, including NGCF [53], LightGCN [18], and NCL [30].

As the task is to predict users' file-level contribution, file-level behavior modeling is the most critical component. Therefore, we use file-level contribution behaviors as the supervision signals as in Eq. (18). For brevity, we use repository identity to refer to the information of which repository a file belongs to. As the baselines do not explicitly leverage the repository identities of files, we encode their repository identities as a categorical feature through one-hot encoding during embedding construction. To ensure fairness of comparison, we incorporate the project-level interaction signals into the user representations by applying multi-hot encoding on the repositories each user has interacted with. All the baseline models use the same pretrained CodeBERT embeddings as CODER to leverage the rich semantic features in the source code.

4.1.4 Evaluation Metrics. Following previous works [18, 19, 53, 78], we choose Mean Reciprocal Rank (MRR@K), Normalized Discounted Cumulative Gain (NDCG@K), Recall@K (Rec@K), and Hit@K as the evaluation metrics.

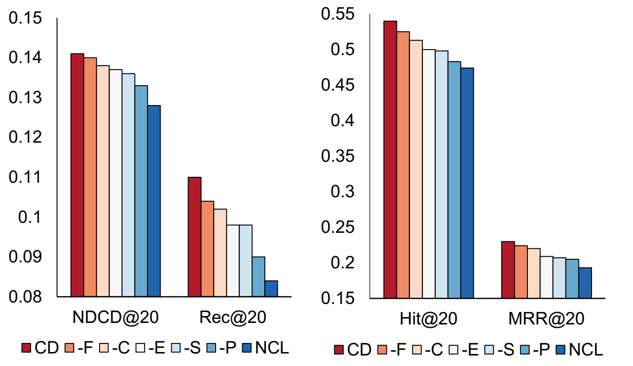

Figure 3: Results among variants of CODER and the best baseline model NCL on the ML dataset.

4.2 Performance

4.2.1 Intra-Project Recommendation. In this setting, we evaluate the model's ability to recommend development tasks under her interacted repositories. For each user u_i, we rank the interactions under repositories they have interacted with in the training set. This setting corresponds to the scenario in which project maintainers recommend new development tasks to existing contributors based on their previous contribution. As shown in Tab. 2, CODER consistently outperforms the baselines by a large margin. On the ML dataset, CODER outperforms the best baseline by 17.9% on NDCG@20, 15.8% on Hit@20, and 8.2% on MRR@20. On the DB dataset, CODER achieves performance improvements of 27.3% on NDCG@20, 29.5% on Hit@20, and 6.0% on MRR@20. Notably, the greatest performance improvement is achieved on the FS dataset, which has the greatest sparsity. CODER achieves a maximum performance improvement of 37.1% on NDCG@5 and 35.6% on NDCG@10. The results show that CODER achieves significant performance improvement over the baseline, and is especially useful when the observed interactions are scarce.

Among the baselines, graph-based method (**G3**) achieves better performances than (**G1**), (**G2**) as they can model the high-order relations between users and items through the interaction graph and the embedding function. LightGCN [18] underperforms NGCF [53] on the DB dataset, whose training set has the greatest density, and outperforms NGCF on the ML and FS datasets. This implies that the message passing scheme of NGCF, which involves multiple linear transformation and non-linear activation, is more effective for denser interactions. Such results justify our design choice in multi-behavioral modeling, which uses the LightGCN propagation scheme. NCL exhibits the strongest performance, demonstrating the importance of the contrastive learning loss in modeling differences among homogeneous types of nodes, which is also included in our model design. Neural-network-based methods (**G2**) generally outperform matrix factorization (**G1**), as they leverage multiple feature transformations to learn the rich semantics in the file embeddings and the user-file interactions.

4.2.2 Cold-Start Recommendation. User contribution is usually sparse due to the considerable workload and voluntary nature of OSS development. In this sense, it is important to accurately capture

Table 2: The overall performance on 3 datasets. The best performance is marked in bold. The second best is underlined.

results among

Dataset	Metric	MF	MLP	NeuMF	NGCF	LightGCN	NCL	CODER	Improvement
ML	NDCG@5	0.065	0.073	0.076	0.091	0.106	0.119	**0.132**	11.2%
	Hit@5	0.162	0.189	0.189	0.237	0.291	0.276	**0.351**	20.5%
	MRR@5	0.098	0.113	0.114	0.137	0.164	0.201	**0.211**	5.0%
	NDCG@10	0.066	0.075	0.081	0.093	0.109	0.118	**0.136**	14.8%
	Hit@10	0.229	0.250	0.263	0.310	0.386	0.337	**0.440**	14.0%
	MRR@10	0.106	0.121	0.124	0.147	0.177	0.209	**0.223**	6.7%
	NDCG@20	0.072	0.081	0.084	0.100	0.116	0.120	**0.141**	17.9%
	Hit@20	0.324	0.343	0.346	0.407	0.457	0.466	**0.540**	15.8%
	MRR@20	0.113	0.127	0.130	0.154	0.185	0.213	**0.230**	8.2%
DB	NDCG@5	0.085	0.079	0.085	0.099	0.082	0.124	**0.160**	29.0%
	Hit@5	0.205	0.191	0.206	0.263	0.237	0.316	**0.390**	23.2%
	MRR@5	0.130	0.118	0.128	0.162	0.132	0.252	**0.260**	3.2%
	NDCG@10	0.086	0.079	0.085	0.100	0.084	0.123	**0.159**	29.4%
	Hit@10	0.267	0.251	0.276	0.361	0.324	0.380	**0.488**	28.4%
	MRR@10	0.138	0.126	0.137	0.175	0.144	0.260	**0.273**	4.9%
	NDCG@20	0.088	0.083	0.088	0.103	0.091	0.125	**0.160**	27.3%
	Hit@20	0.335	0.338	0.362	0.454	0.422	0.437	**0.588**	29.5%
	MRR@20	0.143	0.132	0.143	0.182	0.150	0.264	**0.280**	6.0%
FS	NDCG@5	0.063	0.063	0.067	0.082	0.089	0.106	**0.146**	37.1%
	Hit@5	0.168	0.178	0.179	0.231	0.245	0.283	**0.374**	31.9%
	MRR@5	0.100	0.100	0.107	0.132	0.146	0.170	**0.226**	33.0%
	NDCG@10	0.063	0.065	0.068	0.085	0.092	0.106	**0.144**	35.6%
	Hit@10	0.231	0.244	0.249	0.319	0.332	0.361	**0.467**	29.3%
	MRR@10	0.109	0.110	0.117	0.144	0.157	0.180	**0.239**	32.3%
	NDCG@20	0.067	0.070	0.073	0.090	0.095	0.110	**0.146**	32.7%
	Hit@20	0.307	0.321	0.335	0.406	0.414	0.451	**0.559**	23.9%
	MRR@20	0.114	0.115	0.122	0.150	0.163	0.187	**0.245**	31.4%

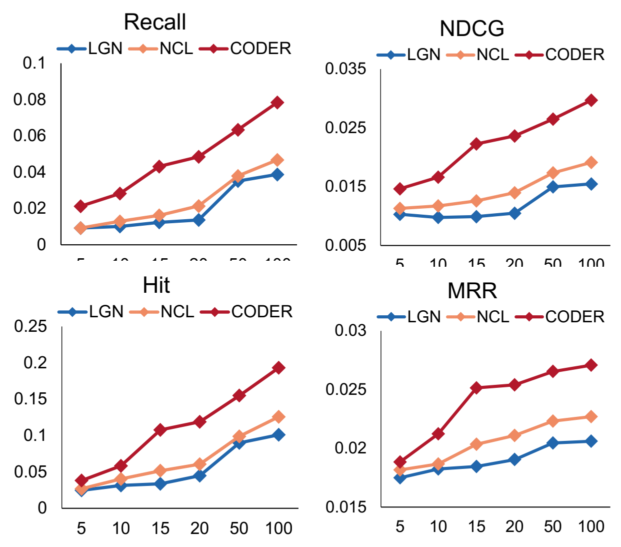

Figure 4: Cross-Project Performance of CODER and the 2 strongest baselines under various $K, K \in [5, 100]$.

the users' preferences with few observed interactions. Our model is thus designed according to this principle. We define cold-start users as users with ≤ 2 interactions in the training set. To evaluate the model performance with fewer interactions, we choose NDCG@K, Recall@K, and Hit@K, where $K \in \{3, 5\}$. The strongest 4 baselines in Tab. 2 are evaluated for comparison.

As observed from Tab. 3, performance for cold-start users is worse than that for all users in Tab. 2. Notably, CODER is able to achieve even greater performance improvement over the baseline models. It can be attributed to the following aspects: 1) CODER learns more accurate representations by fusing the fine-grained semantics of project files with their interacted users, which facilitates the learning of user preferences even in the absence of dense user interactions. 2) By explicitly modeling multiple types of project-level behaviors, CODER effectively models the users' interests to complement the sparse file-level contribution relations, which is more effective than encoding the project-level interactions in the embedding space.

4.2.3 Cross-Project Recommendation. Although 91% developers in our dataset focused on 1 project throughout their development, active contributors can work on multiple projects. For these contributors, the project core team can recommend development tasks based on their development experiences in previous projects.

Table 3: File-level link prediction results for cold-start users. "LGN" stands for the baseline "LightGCN". The best performance is marked in bold. The second best is underlined.

	Metric	NeuMF	NGCF	LGN	NCL	CODER	Impr.
ML	NDCG@3	0.059	0.067	0.068	<u>0.090</u>	**0.126**	40.9%
	Hit@3	0.106	0.123	0.161	<u>0.211</u>	**0.224**	5.9%
	MRR@3	0.081	0.087	0.089	<u>0.119</u>	**0.165**	38.3%
	NDCG@5	0.068	0.078	0.088	<u>0.105</u>	**0.132**	25.8%
	Hit@5	0.161	0.162	0.230	<u>0.261</u>	**0.273**	4.8%
	MRR@5	0.093	0.097	0.105	<u>0.130</u>	**0.177**	36.0%
DB	NDCG@3	<u>0.078</u>	0.063	0.055	0.075	**0.119**	53.0%
	Hit@3	0.152	0.114	0.128	<u>0.165</u>	**0.238**	44.4%
	MRR@3	<u>0.102</u>	0.089	0.070	0.095	**0.157**	54.0%
	NDCG@5	0.086	0.061	0.064	<u>0.086</u>	**0.127**	47.5%
	Hit@5	0.195	0.132	0.165	<u>0.220</u>	**0.287**	30.6%
	MRR@5	<u>0.112</u>	0.093	0.079	0.106	**0.168**	50.2%
FS	NDCG@3	0.079	0.075	0.085	<u>0.092</u>	**0.128**	38.7%
	Hit@3	0.171	0.165	0.179	<u>0.179</u>	**0.242**	35.6%
	MRR@3	0.110	0.095	0.104	<u>0.116</u>	**0.171**	48.0%
	NDCG@5	0.086	0.086	0.085	<u>0.095</u>	**0.137**	44.3%
	Hit@5	<u>0.230</u>	0.220	0.202	0.222	**0.313**	36.2%
	MRR@5	0.124	0.106	0.109	<u>0.125</u>	**0.187**	49.4%

During evaluation, we rank the interactions in projects each user has not yet interacted with in the training set. This setting is considerably more challenging than intra-project recommendation since the candidate item pool is significantly larger. According to the results in Fig. 4, CODER consistently achieves superior performance by a large margin with respect to the baselines, especially for $K \geq 20$. The results show that CODER jointly learns inter-project differences to choose the correct repositories and characterize intra-project distinctions to recommend the correct files within the chosen repositories.

4.2.4 Ablation Studies. In Fig. 3, we compare the performance of our model (abbreviated as CD) among its 5 variants. **CD-F** removes the code-user modality fusion strategy in Eq. (1). **CD-C** excludes the structural contrastive learning objective in Eqs. (22, 23). **CD-E** does not use the pretrained CodeBERT embeddings and instead applies TF-IDF encoding on the source code, a common approach in project recommendation models [62]. **CD-P** removes the project-level aggregation in Sec. 3.2.2. **CD-S** disables the structural-level aggregation in Sec. 3.1.2. Results on the ML dataset are in Fig. 3.

We observe that all 6 variants of CODER outperform NCL, among which the full model (CD) performs the best, indicating the importance of each component in our model design. The performance drops most significantly when we disable project-level aggregation in **CD-P**, indicating the importance of explicitly modeling user-project interactions through graph structures. We also observe a considerable decrease when we remove the structural-level aggregation (**CD-S**), implying that the structural information of files has a significant contribution towards the file representation. **CD-E** does not lead to a more significant performance decrease, but is outperformed by **CD-F** where fine-grained code representations are present. Thus, user behaviors and project structural clues are more important than semantic features in code recommendation.

5 RELATED WORK

5.1 Research in Open Source Code

Open sourcing has grown into a standard practice for software engineering [23] and attract researchers to study social coding [70]. Analytical studies focus on users' motivation [10, 67], expertise [50], collaboration patterns [38], and factors that impact the popularity [2] of projects. Methodological studies explore project classification [74] and code search [33], connecting publications with projects [45]. Recently, the field of Neural Code Intelligence (NCI), which uses deep learning techniques to tackle analytical tasks on source code, has emerged as a promising direction to improve programming efficiency and reducing human errors within the software industry [64]. Although previous works explored the recommendation task in OSS development settings such as automatic suggestions of API function calls [20, 40], Good First Issues [1, 60], and data preparation steps [65], no previous works have studied the challenging task of code recommendation task, which requires in-depth understanding of diverse user-item interactions and OSS projects written in multiple programming languages.

5.2 Recommender Systems

The advances in deep learning greatly facilitate the evolution of recommender systems [3, 17, 41, 44, 72, 73]. Motivated by the success of Graph Neural Networks (GNN) [9, 34, 52, 75, 76, 80, 81], a series of graph-based recommender systems [29, 55, 59] are proposed, which organize user behaviors into heterogeneous interaction graphs. These methods formulate item recommendation as link prediction or representation learning tasks [46, 54, 77], and utilize high-order relationships to infer user preferences, item attributes, and collaborative filtering signals [45, 58, 68]. Noticeably, traditional recommendation models cannot be easily transferred to code recommendation as they do not model unique signals in OSS development, such as project hierarchies and code semantics.

6 CONCLUSION AND FUTURE WORKS

We are the first to formulate the task of code recommendation for open source developers. We propose CODER, a code recommendation model for open source projects written in diverse languages. Currently, our approach only considers recommending existing files to users. As CODER harnesses the metadata and semantic features of files, it cannot deal with users creating new files where such information of the candidate files is absent. We plan to generalize our framework by allowing users to initialize files under their interested subdirectories. Moreover, our source code encoding scheme can be further improved by harnessing knowledge about programming languages such as using Abstract Syntax Tree (AST) [39] and data flow [15, 20] (graphs that represent dependency relation between variables). Our current encoding scheme is a computationally efficient way to deal with the diversity of programming languages. In the future, we plan to incorporate such domain knowledge to improve the file representations at a finer granularity. Our current encoding scheme is a computationally efficient way to deal with the diversity of programming languages. Moreover, the user representations can be further enhanced by modeling users' social relations [6, 36] and behaviors [25, 63, 66].

ACKNOWLEDGMENTS

This work was partially supported by NSF 2211557, NSF 1937599, NSF 2119643, NASA, SRC, Okawa Foundation Grant, Amazon Research Awards, Cisco research grant, Picsart Gifts, and Snapchat Gifts.

REFERENCES

[1] Jan Willem David Alderliesten and Andy Zaidman. 2021. An Initial Exploration of the "Good First Issue" Label for Newcomer Developers. In *CHASE*. 117–118.

[2] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *ICSME*. 334–344.

[3] Jin Chen, Defu Lian, Binbin Jin, Kai Zheng, and Enhong Chen. 2022. Learning Recommenders for Implicit Feedback with Importance Resampling. In *WWW*. 1997–2005.

[4] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology* 122 (2020), 106274.

[5] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES*. 1–10.

[6] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *WWW*. 417–426.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of EMNLP 2020*. 1536–1547.

[8] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *RLGM@ICLR*.

[9] Scott Freitas, Diyi Yang, Srijan Kumar, Hanghang Tong, and Duen Horng Chau. 2022. Graph vulnerability and robustness: A survey. *TKDE* (2022).

[10] Marco Gerosa, Igor Wiese, Bianca Trinkenreich, Georg Link, Gregorio Robles, Christoph Treude, Igor Steinmacher, and Anita Sarma. 2021. The shifting sands of motivation: Revisiting what drives contributors in open source. In *ICSE*. 1046–1058.

[11] GitHub. 2016. *The State of the Octoverse.* https://octoverse.github.com/2016/

[12] GitHub. 2022. *Collection: Programming Languages.* https://github.com/collections/programming-languages

[13] GitHub. 2022. Github Number of Repositories. https://github.com/search.

[14] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*. 249–256.

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *NIPS* 30 (2017).

[17] Junheng Hao, Tong Zhao, Jin Li, Xin Luna Dong, Christos Faloutsos, Yizhou Sun, and Wei Wang. 2020. P-companion: A principled framework for diversified complementary product recommendation. In *CIKM*. 2517–2524.

[18] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *SIGIR*. 639–648.

[19] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *WWW*. 173–182.

[20] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. 2021. Pyart: Python api recommendation in real-time. In *ICSE*. 1634–1645.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. In *IJCAI*. 2269–2275.

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[23] Jyun-Yu Jiang, Pu-Jen Cheng, and Wei Wang. 2017. Open source repository recommendation in social coding. In *SIGIR*. 1173–1176.

[24] Yiqiao Jin, Xiting Wang, Yaru Hao, Yizhou Sun, and Xing Xie. 2023. Prototypical Fine-tuning: Towards Robust Performance Under Varying Data Sizes. In *AAAI*.

[25] Yiqiao Jin, Xiting Wang, Ruichao Yang, Yizhou Sun, Wei Wang, Hao Liao, and Xing Xie. 2022. Towards fine-grained reasoning for fake news detection. In *AAAI*, Vol. 36. 5746–5754.

[26] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*.

[27] Diederik P Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

[28] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. In *ICLR*.

[29] Anchen Li, Bo Yang, Huan Huo, and Farookh Hussain. 2022. Hypercomplex Graph Collaborative Filtering. In *WWW*. 1914–1922.

[30] Zihan Lin, Changxin Tian, Yupeng Hou, and Wayne Xin Zhao. 2022. Improving Graph Collaborative Filtering with Neighborhood-enriched Contrastive Learning. In *WWW*. 2320–2329.

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[32] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. 2016. Hierarchical question-image co-attention for visual question answering. *NIPS* 29 (2016).

[33] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. In *OOPSLA*.

[34] Haitao Mao, Lun Du, Yujia Zheng, Qiang Fu, Zelin Li, Xu Chen, Han Shi, and Dongmei Zhang. 2021. Source free unsupervised graph domain adaptation. *arXiv preprint arXiv:2112.00955* (2021).

[35] Nora McDonald and Sean Goggins. 2013. Performance and participation in open source software on github. In *CHI*. 139–144.

[36] Xin Mei, Xiaoyan Cai, Sen Xu, Wenjie Li, Shirui Pan, and Libin Yang. 2022. Mutually reinforced network embedding: An integrated approach to research paper recommendation. *Expert Systems with Applications* (2022), 117616.

[37] Antonio Valerio Miceli-Barone and Rico Sennrich. 2017. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In *IJCNLP*. 314–319.

[38] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. 2022. Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. *Organization* 1, 2 (2022), 3.

[39] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *SIGSOFT*. 511–522.

[40] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. Focus: A recommender system for mining api function calls and usage patterns. In *ICSE*. 1050–1060.

[41] Sejoon Oh, Ankur Bhardwaj, Jongseok Han, Sungchul Kim, Ryan A Rossi, and Srijan Kumar. 2022. Implicit Session Contexts for Next-Item Recommendations. In *CIKM*. 4364–4368.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *NIPS* 32.

[43] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *UAI*. 452–461.

[44] Walid Shalaby, Sejoon Oh, Amir Afsharinejad, Srijan Kumar, and Xiquan Cui. 2022. M2TRec: Metadata-aware Multi-task Transformer for Large-scale and Cold-start free Session-based Recommendations. In *RecSys*. 573–578.

[45] Huajie Shao, Dachun Sun, Jiahao Wu, Zecheng Zhang, Aston Zhang, Shuochao Yao, Shengzhong Liu, Tianshi Wang, Chao Zhang, and Tarek Abdelzaher. 2020. paper2repo: GitHub repository recommendation for academic papers. In *WWW*. 629–639.

[46] Kartik Sharma, Yeon-Chang Lee, Sivagami Nambi, Aditya Salian, Shlok Shah, Sang-Wook Kim, and Srijan Kumar. 2022. A Survey of Graph Neural Networks for Social Recommender Systems. *arXiv preprint arXiv:2212.04481* (2022).

[47] Nitish Srivastava and Russ R Salakhutdinov. 2012. Multimodal learning with deep boltzmann machines. *NIPS*.

[48] Igor Steinmacher, Ana Paula Chaves, Tayana Uchoa Conte, and Marco Aurélio Gerosa. 2014. Preliminary empirical identification of barriers faced by newcomers to Open Source Software projects. In *SBES*. 51–60.

[49] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.

[50] Rahul Venkataramani, Atul Gupta, Allahbaksh Asadullah, Basavaraju Muddu, and Vasudev Bhat. 2013. Discovery of technical expertise from open source code repositories. In *WWW*. 97–98.

[51] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *ASE*. 397–407.

[52] Haonan Wang, Jieyu Zhang, Qi Zhu, and Wei Huang. 2022. Augmentation-free graph contrastive learning. *arXiv preprint arXiv:2204.04874* (2022).

[53] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural graph collaborative filtering. In *SIGIR*. 165–174.

[54] Xiang Wang, Tinglin Huang, Dingxian Wang, Yancheng Yuan, Zhenguang Liu, Xiangnan He, and Tat-Seng Chua. 2021. Learning intents behind interactions with knowledge graph for recommendation. In *WWW*. 878–887.

[55] Xiting Wang, Kunpeng Liu, Dongjie Wang, Le Wu, Yanjie Fu, and Xing Xie. 2022. Multi-level recommendation reasoning over knowledge graphs with reinforcement learning. In *WWW*. 2098–2108.

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *EMNLP*. 38–45.

[57] Junfei Wu, Weizhi Xu, Qiang Liu, Shu Wu, and Liang Wang. 2022. Adversarial Contrastive Learning for Evidence-aware Fake News Detection with Graph Neural Networks. *arXiv preprint arXiv:2210.05498* (2022).
[58] Le Wu, Peijie Sun, Yanjie Fu, Richang Hong, Xiting Wang, and Meng Wang. 2019. A neural influence diffusion model for social recommendation. In *SIGIR*. 235–244.
[59] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. 2019. Session-based recommendation with graph neural networks. In *AAAI*, Vol. 33. 346–353.
[60] Wenxin Xiao, Hao He, Weiwei Xu, Xin Tan, Jinhao Dong, and Minghui Zhou. 2022. Recommending good first issues in GitHub OSS projects. In *ICSE*. 1830–1842.
[61] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks?. In *ICLR*.
[62] Wenyuan Xu, Xiaobing Sun, Xin Xia, and Xiang Chen. 2017. Scalable relevant project recommendation on GitHub. In *Internetware*. 1–10.
[63] Weizhi Xu, Junfei Wu, Qiang Liu, Shu Wu, and Liang Wang. 2022. Mining Fine-grained Semantics via Graph Neural Networks for Evidence-based Fake News Detection. *arXiv preprint arXiv:2201.06885* (2022).
[64] Yichen Xu and Yanqiao Zhu. 2022. A Survey on Pretrained Language Models for Neural Code Intelligence. *arXiv.org* (Dec. 2022). arXiv:2212.10079v1 [cs.SE]
[65] Cong Yan and Yeye He. 2020. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *SIGMOD*. 1539–1554.
[66] Ruichao Yang, Xiting Wang, Yiqiao Jin, Chaozhuo Li, Jianxun Lian, and Xing Xie. 2022. Reinforcement Subgraph Reasoning for Fake News Detection. In *KDD*. 2253–2262.
[67] Yunwen Ye and Kouichi Kishida. 2003. Toward an understanding of the motivation of open source software developers. In *ICSE*. 419–429.
[68] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *KDD*. 974–983.
[69] Xueli Yu, Weizhi Xu, Zeyu Cui, Shu Wu, and Liang Wang. 2021. Graph-based Hierarchical Relevance Matching Signals for Ad-hoc Retrieval. In *WWW*. 778–787.
[70] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218.
[71] Zhou Yu, Jun Yu, Yuhao Cui, Dacheng Tao, and Qi Tian. 2019. Deep modular co-attention networks for visual question answering. In *CVPR*. 6281–6290.
[72] Fajie Yuan, Xiangnan He, Haochuan Jiang, Guibing Guo, Jian Xiong, Zhezhao Xu, and Yilin Xiong. 2020. Future data helps training: Modeling future contexts for session-based recommendation. In *WWW*. 303–313.
[73] Jinghao Zhang, Yanqiao Zhu, Qiang Liu, Shu Wu, Shuhui Wang, and Liang Wang. 2021. Mining Latent Structures for Multimedia Recommendation. In *ACM MM*. 3872–3880.
[74] Yu Zhang, Frank F Xu, Sha Li, Yu Meng, Xuan Wang, Qi Li, and Jiawei Han. 2019. Higitclass: Keyword-driven hierarchical classification of github repositories. In *ICDM*. 876–885.
[75] Jianan Zhao, Xiao Wang, Chuan Shi, Binbin Hu, Guojie Song, and Yanfang Ye. 2021. Heterogeneous graph structure learning for graph neural networks. In *AAAI*, Vol. 35. 4697–4705.
[76] Jianan Zhao, Qianlong Wen, Shiyu Sun, Yanfang Ye, and Chuxu Zhang. 2021. Multi-view Self-supervised Heterogeneous Graph Embedding. In *ECML-PKDD*. 319–334.
[77] Yu Zheng, Chen Gao, Liang Chen, Depeng Jin, and Yong Li. 2021. DGCN: Diversified Recommendation with Graph Convolutional Networks. In *WWW*. 401–412.
[78] Yu Zheng, Chen Gao, Xiang Li, Xiangnan He, Yong Li, and Depeng Jin. 2021. Disentangling user interest and conformity for recommendation with causal embedding. In *WWW*. 2980–2991.
[79] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2014. Patterns of folder use and project popularity: A case study of GitHub repositories. In *ESEM*. 1–4.
[80] Yanqiao Zhu, Weizhi Xu, Jinghao Zhang, Qiang Liu, Shu Wu, and Liang Wang. 2021. Deep graph structure learning for robust representations: A survey. *arXiv preprint arXiv:2103.03036* (2021).
[81] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, and Liang Wang. 2021. Graph contrastive learning with adaptive augmentation. In *WWW*. 2069–2080.

A DATASETS

We categorized the open source projects in our dataset using their GitHub topics. The topics used in each dataset are shown in Tab. 4.

B IMPLEMENTATION DETAILS

To derive **C** in Sec. 3.1.1, we first convert the source code into a set of tokens, then partition them into code segments with equal number of tokens. As the source code is first tokenized and then partitioned at the token level, our partitioning scheme will not partition meaningful tokens in the middle. For example, if "`def`" is a meaningful token, our method will make sure that the token will not be split into multiple parts such as "`de`" and "`f`".

Next, we encode the tokens using CodeBERT. Each token produces a token-level representation. For each code segment c_i, we perform max pooling on all of its token representations to derive $\mathbf{c}_i$. We use CodeBERT to encode files written in all programming languages (PL) as well as natural language (NL), as CodeBERT is able to generalize to a wide range of programming languages and shows outstanding performances on NL-PL tasks. As CodeBERT learns general-purpose code representations for both natural language (NL) and programming languages (PL), CodeBERT is suitable for encoding the rich natural language clues in OSS projects including the `README` file, which contains frequent context-switching between NL and PL.

Table 4: GitHub topics used during the construction of each dataset.

Dataset	Topics
ML	`computer-vision`, `data-science`, `0deep-learning`, `machine-learning`, `neural-network`, `pytorch`, `tensorflow`
FS	`angular`, `css`, `html`, `javascript`, `js`, `nodejs`, `react`, `reactjs`, `vue`, `vuejs`
DB	`database`, `graphql`, `mongodb`, `mysql`, `sql`