IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2018.DOI

Recommending GitHub Projects for Developer Onboarding

CHAO LIU[1,2], DAN YANG[2], XIAOHONG ZHANG[2], BAISHAKHI RAY[3], AND MD MASUDUR RAHMAN[4]

[1]Key Laboratory of Dependable Service Computing in Cyber Physical Society Ministry of Education, Chongqing University, Chongqing, China
[2]School of Big Data & Software Engineering, Chongqing University, China (e-mail: {liu.chao, dyang, xhongz}@cqu.edu.cn)
[3]Department of Computer Science, Columbia University, USA (e-mail: {rayb}@cs.columbia.edu)
[4]Department of Computer Science, University of Virginia, USA (e-mail: {masud}@virginia.edu)

Corresponding author: Dan Yang (e-mail: dyang@cqu.edu.cn).

ABSTRACT Open source platform (e.g., GitHub) creates a tremendous opportunity for developers to learn and build experience. Contribution to open source can be rewarding for developers and advocates the evolutionary progress of the open source software. However, finding a suitable project to contribute can be intimidating for developers because of the enormous possible choices. Due to various social and technical barriers, developers might fail to contribute successfully. Frequent unsuccessful onboarding hampers not only developers' individual advancement but also the evolutionary progress of open source projects. To mitigate developers' costly efforts for onboarding, we propose a learning-to-rank model NNLRank (Neural Network for List-wise Ranking) to recommend projects that developers are likely to contribute. NNLRank leverages project features and developers' experience to recommend projects for onboarding. We develop an efficient approach to optimize the neural network where we leverage a list-wise loss function which intends to minimize the difference between the predicted projects list and the ground-truth list preferred by developers. We evaluate NNLRank with 2044 successful onboarding decisions from GitHub and compare it with three standard learning-to-rank models and a prior onboarding tool. Experimental results show that NNLRank can provide effective and efficient onboarding recommendation to developers, substantially outperforming the previous models.

INDEX TERMS Developer Onboarding, Recommender System, Learning to Rank

I. INTRODUCTION

Open Source Software (OSS) ecosystems like GitHub and BitBucket provide developers shared and convenient technical platforms for social interactions, technical collaborations, and reputation buildings [1]–[4]. Thus, a large number of professional and amateur developers get attracted to work on OSS ecosystems (i.e. onboarding) to pursue their interests and goals [5], [6]. For a thriving OSS ecosystem, successful onboarding becomes a driving force. However, due to various technical and social barriers [1], [7], [8] developers often face difficulties in successful onboarding—finding a suitable project to join and contribute many commits [9]. How can I find a good open source project to join? a developer asked in Stack Exchange [10]. Similar questions are asked in other forums [11].

In the current scenario, developers often give significant effort (mostly manual) to find suitable open source projects [12], [13] to contribute. They spend a significant amount of time to learn those projects [14], [15]. They often consult different resources such as project documentation, mailing lists, forums, bug tracking system, etc. to learn the projects of interest.

Even after such manual effort when developers start contributing code, it is not guaranteed that their code will be accepted. Developers start to write code (e.g., bug fix patch, new feature) and commit in the hope that their contributions will be accepted. Before accepting contributions (e.g., commits) the core project members often verify it with their project relevancy. If the contributions remain unsatisfactory, developers have to rewrite, review and test new commits again and again until they pass the project standard and goal. This process can be very tedious due to incorrect initial project selection. The situation can be even worse for newbies who have less experience with the open source platform, consequently leads to frustration and eventual quitting [7], [8].

From the projects' perspective, an unsuccessful onboarding attempt impedes the overall development

progress—frequently failed onboarding means the core project members have to spend significant efforts to review unsatisfactory commits submitted during onboarding [7], [8]. Thus, the overall project growth can be delayed with enormous frustrations from developers including core members.

Practitioners provide various resources including documentation (e.g., [16]), platforms (CodeTriage [17], OpenHatch [18]) to ease the manual effort of developers. Researchers also build tool support to help newbies in the onboarding process [19]–[21]. But none of these resources directly help developers to recommend relevant projects based on their expertise and previous work history. Recently, GitHub provides project recommendation [22] to its users mostly based on their stars and the people they follow. However, such a recommendation is more suitable for generic project usage than developer onboarding.

Previously, many models have been proposed to recommend items to work on for newcomers, such as source code [23], code relevant items [24], [25], etc. However, these works presume that the developer already joined in a project.

In contrast, researchers have identified some factors, such as technical background, collaboration history, and project growth, that may affect project level onboarding. However, recommending projects for onboarding is still challenging because of the complex interactions between multiple social and technical factors [8] affecting the onboarding process.

Proposed Approach. To capture the complex onboarding pattern, we analyze different project features and activity & expertise of developers. Leveraging these features, we propose a learning-to-rank model named NNLRank (Neural Network for List-wise Ranking) to recommend projects that developers are likely to contribute. NNLRank learns a ranking function (represented by a neural network [26]) to score a list of candidate projects, where the top-n projects are recommended to developers. To iteratively optimize the network, we apply a list-wise ranking loss function that aims to minimize the difference between the predicted list (scores of projects) and the ground-truth list preferred by developers. We leverage a learning rate decay method to enable the fast convergence of model optimization. NNLRank can further handle a list of candidate projects simultaneously which enables faster onboarding processing.

We verify NNLRank by investigating 2044 successful onboarding decisions from GitHub developers. Each decision involves 1428 different candidate projects on average with a total of 2.9 million instances. To confirm the effectiveness of NNLRank over standard learning-to-rank models, we evaluate the performance of SVMRank [27], BPNet [28], and SVM [29] for the project onboarding task, where BPNet and SVM are 2 models for solving the ranking problem via classification. We further compare our NNLRank with a previously proposed link prediction based tool, LP [12]. We use 5-fold cross-validation and evaluate the performance of these models by standard evaluation metrics: MRR (Mean Reciprocal Rank), MAP (Mean Average Precision), and Recall in different cutoffs (i.e. 10, 20).

Experimental results show that NNLRank achieves a performance of 0.462, 0.454, 0.457 at MRR, MAP@10, and Recall@10 respectively; and it significantly outperforms all other models. In particular, compared with the second best model SVMRank, our NNLRank achieves a percentage improvement of 28.69%, 29.34%, and 29.10% at MRR, MAP@10, and MAP@20 respectively. Moreover, NNLRank also works efficiently with 20.43 seconds to train and test for the cross-validation.

In summary, we make the following contributions:

- We build a recommender system, NNLRank, for the GitHub projects onboarding.
- Design nine features to capture developers' onboarding pattern.
- Analyze the features that impact the model's accuracy.
- Evaluate the model empirically w.r.t. prediction accuracy and run-time performance.

Paper Organization. The remainder of this paper is organized as follows. The background and research questions are respectively presented in Sections II and III. The methodology and experimental setup are detailed in Sections IV and V respectively, followed by the results, discussion, and implication in Sections VI, VII, and VIII. And the threats to validity and the conclusion are respectively described in Sections IX and X.

II. BACKGROUND

Researchers urge an onboarding tool to mitigate developers' searching efforts for projects with less technical and social barriers [30], [31]. Some existing tools can help developers overcome obstacles in projects by supplying a search engine for API usages and high-quality examples [32], predicting code changing requests [32], recommending code relevant items such as closely related code, problem reports, newsgroup articles and etc. [24], [25], [33], [34] However, these tools hypothesize developers have already decided to join in a project, but making such a decision is not easy [1], [7], [8]. Therefore, it is necessary to build a recommender system to recommend or filter projects (associated with developers' search terms) in the first place. Based on developers' social connections among projects, Matek et al. [12] developed an unsupervised model called LP (Link Prediction) to identify developers' participation behaviors instead of their onboarding choices. By contrast, in this paper, we address the onboarding problem and propose a recommendation tool which helps developers to find suitable projects to contribute.

Learning to Rank. Collaborative filtering and content-based filtering are two widely used recommender systems [35], [36]. However, they cannot be directly applied to underlining onboarding task, since both model types require developers' preference degrees on the onboarded projects which are not available in our dataset.

An alternative solution is the learning to rank, which turns the recommendation problem into a ranking problem by constructing a scoring model, where top-n scored projects are recommended. In general, 3 types of learning-to-rank models can be used [37], [38]:

1) Point-wise ranking: it transforms a ranking problem into a classification problem, learning the probability that an instance (e.g., A, B, or C) occurs one by one [37]. For our scenario, a classifier (e.g., SVM [29] and BPNet [28]) predicts developers' probability to onboard each project (A, B, or C), where the projects with top-n probabilities are recommended. However, point-wise models may fail to work as they ignore local relationships between projects so that projects with similar features are all assigned with the same scores [37].

2) Pair-wise ranking: it also turns to be a classification problem, but learning the sequential relations between instances (e.g., A>B and B>C). Knowing such relations between projects, candidate projects can easily be sorted and recommended for developers. SVMRank [27] is a representative model, but pair-wise ranking has a high computation complexity since all pairs of projects are usually required to be processed.

3) List-wise ranking: it directly learns the sequential relations among a list of instances just one time (e.g., A>B>C). It is designed to reduce the computation complexity by processing a list of projects only once, in a more natural and straightforward fashion [37], instead of transforming a given list of projects into pairs or individuals [39]. Usually, the list-wise ranking model is a neural network based model [39]. Its essential factor is optimization of the network via an effective loss function, which is generally calculated as a difference between the predicted ranking $\boldsymbol{x}=\{x_i\}_{i=1}^n$ and the ground-truth ranking $\boldsymbol{y}=\{y_i\}_{i=1}^n$. Cross-entropy [40] and cosine similarity [41] are two typical loss functions. However, it has been analyzed that both these loss functions are not sound, namely inadequate to minimize the difference between the predicted and ground-truth rankings in some cases. On the other hand, the cosine similarity is not convex, which implies the trained model is likely to have poor performance in testing data. We adapt a likelihood loss function satisfying the soundness and convexity [39] $L(\boldsymbol{x},\boldsymbol{y})=\prod_{i=1}^{n}\frac{exp(x_i)}{\sum_{k=1}^{n}exp(x_i)}$, where the onboarded project in $\boldsymbol{x}$ is put to the first place (x_1) according to the ground truth $\boldsymbol{y}$. Due to its theoretical validity and experimental performance, this likelihood loss function has been widely used.

III. RESEARCH QUESTIONS

In an open source ecosystem like GitHub, a successful developer onboarding occurs when a developer joins a new project and contributes many subsequent commits.

Based on the above definition of onboarding, we develop a tool which leverages the abundant histories of developers' activities in OSS projects, as hosted in GitHub. Our main goal is to investigate how effectively and efficiently our tool can recommend relevant projects for onboarding. We also analyze how different factors impact the onboarding performance.

In particular, we aim to investigate:

RQ1. How accurately can NNLRank recommend onboarding to GitHub projects?

Here, we check the recommendation accuracy of our model and compare it with previously proposed models. Intuitively, the proposed model provides a better recommendation for developers with more historical information. As the amount of historical information impacts the recommendation performance, we further investigate the prediction accuracy of our model for developers with different prior experience.

RQ2. What are the most important features that influence onboarding recommendation?

We explore the relative importance of extracted features on the performance of our model via a sensitivity analysis. Additionally, we analyze the impact of domain features (e.g., functional project similarity) on onboarding decision.

RQ3. What is the run-time overhead of NNLRank?

The goal of this RQ is to investigate the run-time performance of our model. In particular, we record and analyze how long does our model take to train and test.

IV. METHODOLOGY: NNLRANK MODEL

To recommend suitable projects for a developer to onboard, we build a list-wise ranking model called NNLRank. NNLRank leverages a neural network as a ranking function, which can process a list of candidate projects simultaneously. It takes 9 features extracted from projects and developers' profile information as input and predicts the preference scores for projects that developers would like to onboard. Then, sorting the predicted scores in descending order NNLRank recommends top-n projects for developer onboarding. However, the performance of NNLRank is affected by its neural network structure, loss function, and network optimization approach. Thus, in this section, we discuss our NNLRank model implementation in details and evaluate how these components affect the NNLRank in Section VII.

A. FEATURE SELECTION

Generally, it is assumed that developers would like to work with their frequently collaborated developers, and join the projects that match their expertises and preferences [1], [42]. To predict the developer's preference score on a candidate project, NNRank leverages 9 features including developer's social tie to the project's owner, the language expertise matching with the project's requirement, number of current commits in the project, number of project members coming from the same company as the developer, and the durations from developer's onboarding time to the project creation time, to the first/last membership time, and to the first/last commit time. We discuss in details how we measure these features in Section V-B.

B. NEURAL NETWORK.

Figure 1 shows that the network structure of our model contains 4 layers with two hidden layers, one input, and one output layer. The input layer has 9 nodes, corresponding to

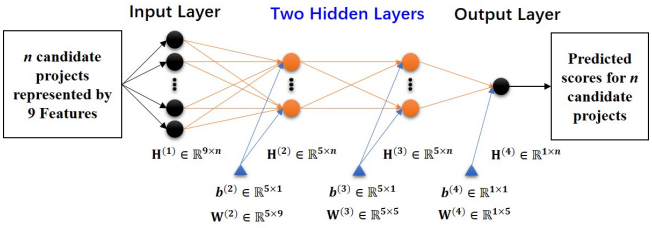


FIGURE 1: **Neural network structure.**

9 designed features; each hidden layer contains 5 nodes, and the output layer only contains 1 node for the score.

Precisely, to predict the preference scores of n candidate projects represented by 9 features, the network takes all these projects $X \in \mathbb{R}^{9\times n}$ for its input layer $H^{(1)}=X$, and passes them through each layer from left to right. For the m-th layer ($m \in [2,4]$), its output is:

$$H^{(m)} = g(W^{(m)}H^{(m-1)} + \boldsymbol{b}^{(m)}\mathbf{1}^{(m)}) \tag{1}$$

where $W^{(m)}$ and $\boldsymbol{b}^{(m)}$ are the weights and biases of the m-th layer, respectively; $\mathbf{1}^{(1)} \in \mathbb{R}^{1\times n}$ is a vector with all elements equaling to one for reshaping the biases; and g is an Arctan activation function, $g(x)=atan(x)$, whose impact is investigated in Section VII by comparing it with other common activation functions; the last layer $H^{(4)}$ outputs the scores for n candidate projects.

Note that, as the weights and biases are unknown at first, we initialize them with a simple and prevalent method [43], [44]. In specific, for the m-th layer, all elements in the bias $\boldsymbol{b}^{(m)}$ are set to zero, while the weight $W^{(m)}$ is initialized by the uniform random numbers ranging from $-\sqrt{6}(q^{(m)} + q^{(m-1)})^{-\frac{1}{2}}$ to $\sqrt{6}(q^{(m)} + q^{(m-1)})^{-\frac{1}{2}}$, where $q^{(m)}$ is the number of nodes in the m-th layer. Section VII investigates how the randomness affects the model performance.

C. LOSS FUNCTION.

To optimize weights and biases in the network, we leverage a list-wise ranking loss function which aims to minimize the difference between the predicted score list and the ground-truth list, namely assigning the highest score to the successfully onboarded project.

Specifically, for a list of candidate projects $X=\{\boldsymbol{x}_i\}_{i=1}^n$ with the onboarded project at the first place, their scores predicted by the neural network (f) can be written as $f(X)=\{f(\boldsymbol{x}_i)\}_{i=1}^n$. Then, the probability of this ideal permutation can be expressed as [39], $p(f(X))=\prod_{i=1}^n \frac{exp(f(\boldsymbol{x}_i))}{\sum_{k=i}^n exp(f(\boldsymbol{x}_k))}$. Furthermore, our loss function aims to maximize $p(f(X))$, the probability of ranking the onboarded project at the first place. Moreover, the loss function can also be transformed to minimize $-log(p(f(X)))$. To efficiently solve this optimization problem, we further transform the loss function as Eq. (2).

$$-log\left(p(f(X))\right) \geq -\sum_{i=1}^{n} f(\boldsymbol{x}_i) + \sum_{i=1}^{n}\sum_{k=i}^{n} f(\boldsymbol{x}_k) = \sum_{i=2}^{n}(i-1)f(\boldsymbol{x}_i) \tag{2}$$

Intuitively, the transformed loss function aims to minimize the weighted sum of the scores without the onboarded project, where the i-th project (x_i) is assigned to a higher weight (i-1). Finally, the loss function is written as:

$$\arg\min_{W,b} L = \sum_{i=2}^{n}(i-1)f(\boldsymbol{x}_i) + \frac{\lambda}{2}\sum_{m=1}^{M}\left(\|W^{(m)}\|_F^2 + \|b^{(m)}\|_2^2\right) \tag{3}$$

where λ is a regularization parameter to control the complexity of weights and biases. λ ranges from 0 to 1, and a higher value indicates a lower complexity of weights and biases. It is usually assumed that lower complexity is beneficial to the network optimization, which is also confirmed by our analysis in Section VII. Therefore, we set λ to 1 as default.

D. NETWORK OPTIMIZATION.

With the help of the above loss function, we input lists of candidate projects (i.e. training data) to the network, and iteratively optimize the weights and biases in the network. Generally, for the i-th list of projects, we calculate a value L_i for the loss function. We perform a gradient descent method to optimize parameters in the network. The optimization process stops when the loss function value converges to a small threshold (ε). We set ε=0.1 to ensure the value of loss function is stable enough when the training is stopped. Otherwise, the model stops training when all lists of training data have been used.

The critical part of the network optimization is the gradient descent method. It first computes the gradients of the loss function and then updates the weights and biases in each layer. Particularly, the gradients of the loss function L are computed as Eq. (4), where $H_i^{(m-1)}$ is the output of the (m-1)-th layer of the i-th candidate project in the list.

$$\begin{aligned}\frac{\partial L}{\partial W^{(m)}} &= \sum_{i=2}^{n}\boldsymbol{\Delta}_{i=2}^{(m)}\boldsymbol{H}_i^{(m-1)^T} + \lambda\boldsymbol{W}^{(m)} \\ \frac{\partial L}{\partial \boldsymbol{b}^{(m)}} &= \sum_{i=2}^{m}\boldsymbol{\Delta}_i^{(m)} + \lambda\boldsymbol{b}^{(m)}\end{aligned} \tag{4}$$

For layer $m \in [1,M]$, the updating equation $\boldsymbol{\Delta}_i^{(m)}$ is calculated as Eq. (5), where the operation $\odot$ denotes the element-wise multiplication; g' is the derivative of the activation function g; and $\boldsymbol{z}_i^{(m+1)} \triangleq W^{(m)}H_i^{(m-1)} + \boldsymbol{b}^{(m)}$.

$$\boldsymbol{\Delta}_i^{(m)} = \begin{cases} (i-1)g'(\boldsymbol{z}_i^{(m)}), & m = M \\ (W^{(m+1)^T}\boldsymbol{\Delta}_i^{(m+1)}) \odot g'(\boldsymbol{z}_i^{(m)}), & m < M \end{cases} \tag{5}$$

We update the weights and biases by Eq. (6) until it converges, where μ is the learning rate (μ=10^{-2} in default).

$$W^{(m)} = W^{(m)} - \mu\frac{\partial L}{\partial W^{(m)}}, \boldsymbol{b}^{(m)} = \boldsymbol{b}^{(m)} - \mu\frac{\partial L}{\partial \boldsymbol{b}^{(m)}} \tag{6}$$

To accelerate the model training, we let the model to learn quickly at the beginning with a large learning rate ($\mu = 0.01$). Then, we gradually decrease learning rate as μ_t=$10^{-\lceil t/10 - t_{after}\rceil}\mu_{t-1}$ when the iteration number t increases, where $\lceil t/10 - t_{after}\rceil$ is an upward rounding

operation on $t/10 - t_{after}$; and the learning rate begins to decay from the iteration t_{after}=0.

Algorithm 1 summarizes the details of model training procedures. Note that, we empirically initialize 3 coefficients $(\varepsilon, \mu, t_{after})$ for network optimization. We investigate their impacts in details on the model performance in Section VII.

Algorithm 1: NNLRank

Input: Training set $\{X_i\}_{i=1}^{n}$, number of network layers $M+1$, learning rate μ, total iteration number I_t, the initial step for learning rate decay t_{after}, regularization coefficient λ, and convergence error ε.
Output: Weights and biases $\{W^{(m)}, \boldsymbol{b}^{(m)}\}_{m=1}^{M}$.
// Initialization:
Initialize $W^{(m)}$ and $\boldsymbol{b}^{(m)}$ for all layers, $m \in [1, M]$.
// Optimization by the gradient descend method:
For $t = 1, 2, \ldots, I_t$
  Randomly select a list of candidate projects X_t
  Set $H^{(1)} = X_t$
  // Forward propagation:
  For $m = 2, 3, \ldots, M+1$
    Do forward propagation to get $H^{(m)}$ as Eq. (1)
  End
  // Compute gradients:
  For $m = M, M-1, \ldots, 1$
    Obtain gradients according to Eq. (4)
  End
  // Back Propagation:
  Update μ by learning rate decay method.
  For $m = 1, 2, \ldots, M$
    Update $W^{(m)}$ and $\boldsymbol{b}^{(m)}$ as Eq. (6)
  End
  // Stopping criteria:
  Calculate the loss function L_t at the t-th iteration as Eq. (3).
  If $t > 1$ and $|L_t - L_{t-1}| < \varepsilon$
    Go to **Return**
  End
End
Return: $\{W^{(m)}, \boldsymbol{b}^{(m)}\}_{m=1}^{M}$

V. EXPERIMENTAL SETUP

A. DATA COLLECTION

We now describe the projects we studied, and how we gathered and analyzed the data.

Sampling Projects. We investigated 2-years onboarding decisions (2012-2013) from a GHTorrent dump[1], whose projects are written in 8 popular general purpose programming languages (Javascript, Java, Python, PHP, Ruby, C++, C, and C#)[2] on GitHub.

It is well known that GitHub contains numerous inactive and personal projects that cannot be joined by other developers, which might lead to biased analysis [1], [45]. Therefore, we removed the deleted, private, and forked repositories; excluded projects that do not have at least 2 watchers and 5 developers. Moreover, projects with non-English and insufficient description tend to be personal or toy repositories, and they can hardly attract worldwide developers to join. Thus, we removed the projects that have non-English words in the description, and the projects do not have at least 5 words (median of the collected projects) in the description. In total, we sampled 2,540 active projects from GitHub.

[1]GHTorrent dump: a mysql dump with version number 20161101 downloaded from http://ghtorrent.org/downloads.html.

[2]Top languages in GitHub: http://githut.info/, accessed April 2018

Sampling Developers. The sampled 2,540 projects contain 27,019 developers. As a research with inactive GitHub users would lead to biased conclusions on their onboarding behaviors [1], our study thus only focuses on the active developers in terms of their profiles and we excluded the deleted users, fake users, and users without geographical information. The fake users are the accounts who spam other websites using GitHub authentication, and they are identified by the GitHub and labeled in the collected dataset. In this way, we obtained 8,374 active developers who relate to 46,302 repositories on GitHub.

Sampling Successful Onboarding. The sampled 2,540 active projects contain 463,116 onboarding decisions (i.e. membership information recorded in GHTorrent). As developers come and go frequently for projects [33], a successful onboarding should include significant commits after a developer joins a project. In this study, we chose the onboarding decision with at least 6 commits (median of the sampled active projects) as a successful onboarding. Then we obtained 2,142 successful decisions. Additionally, as developers cannot join the projects created by themselves, we removed 98 decisions of such cases. Finally, we collected 2,044 successful onboarding decisions.

Study Subject. In summary, we studied a total of 2,044 successful onboarding decisions associated with 1,070 developers and 1,672 open source GitHub projects; 14.95% of these developers made multiple onboarding decisions (Table 2); and most (79.16%) of developers joined at least one project prior to onboarding a new project (Table 3). To predict these decisions, we extract project features from projects and developers' prior experience, which involves 132,295 repositories and 58,259 developers in total. Details show in Table 1.

TABLE 1: **Statistics of Collected Onboarding Dataset.**

Type	Name	Count
Onboarding	Successful Decision	2,044
	Developers	1,070
	Projects	1,672
Feature Extraction	Developers	58,259
	Projects	132,295
Data Collection	Developers	27,019
	Projects	2,540

For each onboarding decision, we considered all projects created before the onboarding time as the candidates. Thus the number of candidate projects varies for different decisions and each onboarding decision involves 257 to 2794 candidate projects. Moreover, each decision relates to an average of 1,428 instances (i.e., candidate projects). Therefore, we studied 2,918,832 different instances in total, as the features of an instance are changing at different onboarding time.

B. FEATURE EXTRACTION

We extract 9 features from projects and developers' prior experience to capture developers' onboarding pattern, which

TABLE 2: **Statistics of developers onboarded multiple projects, where a row indicates that there are #Dev number (%Dev percentage) of developers made #Dec number of onboarding decisions, among 2,044 studied onboarding decisions.**

#Dec	#Dev / %Dev	#Dec	#Dev / %Dev
1	1422 / 85.05%	6	3 / 00.18%
2	194 / 11.60%	7	2 / 00.12%
3	31 / 01.85%	12	1 / 00.06%
4	16 / 00.96%	13	1 / 00.06%
5	1 / 00.06%	15	1 / 00.06%

TABLE 3: **Statistics of projects joined by developers before onboarding a new project, where a column indicates that there are #Dec number (%Dec percentage) of decisions are made with the #Pro number of prior joined projects, and a #Pro value corresponds to the #Perc percentile of projects joined by studied 1,070 developers.**

#Pro (Perc)	0 (0)	1 (25)	4 (50)	11 (75)	298 (100)
#Dec	426	270	107	47	1
%Dec	20.84%	13.21%	5.23%	2.30%	0.05%

are summarized in Table 4. We now describe feature extraction in details.

1) Social Tie. It has been observed that developers are willing to join in a project whose members frequently collaborated with them in the past [1] and the project owner is the main reason to attract developers' participations [42]. Thus, we measure the social tie of a developer to a project by the number of collaborated projects with the owner. However, the individual collaboration could be limited for larger projects with many members, as developers are less likely to communicate with the project owner [1]. Thus, the collaboration strength between owner and target developer is measured by the reciprocal of the population in that project. In this way, the final social tie with the project owner equals the reciprocal sum of the member count in each collaborated project. Specifically, let the owner of a project has previously collaborated with a target developer in P_t number of projects before the onboarding time t, where the i-th prior collaborated project $i \in [1, P_t]$ contains $m_{i,t}$ number of members at that time. Then the social tie (Tie_{owner}) between the target developer and the project owner is measured as $Tie_{owner} = \sum_{i=1}^{P_t} \frac{1}{m_{i,t}}$.

2) Technical Ability. Developers are more likely to code in languages that they have used before [1], [42]. To capture this, we use prior language experience ($Tech_{lang}$) to measure how the expertises of developers match the requirement of a target project. For example, if a developer has joined 2 Java projects and 3 PHP projects, and a candidate project is written in Java, then $Tech_{lang}$=2 for this candidate project.

3) Project Growth. A fast-evolving project with more commits often attracts more developers [1], because such a project needs more contribution. We measure the project growth by the number of commits ($Info_{cmt}$) at onboarding time.

TABLE 4: **Extracted Project Features.**

Category	Feature	Description
Social Tie	Tie_{owner}	The reciprocal sum of member counts in each project previously collaborated with the project owner of a candidate project.
Technical Ability	$Tech_{lang}$	Number of the developer's prior joined projects whose language matches the candidate project.
Project Growth	$Info_{cmt}$	Number of current commits in the candidate project.
User Profile	$User_{company}$	Number of candidate project members who are company colleagues of the developer.
Project Time	$Time_{create}$	From onboarding to project creation time.
	$Time_{meb0}$	From onboarding to first member join time.
	$Time_{meb1}$	From onboarding to last member join time.
	$Time_{cmt0}$	From onboarding to first commit time.
	$Time_{cmt1}$	From onboarding to last commit time.

4) User Profile. We assume that a developer would like to join a project whose members are from the same company. Therefore, the developer can quickly adapt to the joined project. In particular, we count the number of members who shared the same company profile ($User_{company}$) at the onboarding time.

5) Project Creation Time. It is discovered that a project in the early lifetime attracts more developers to join in because developers would face more technical and social barriers when a project is stable [1], [42]. We thus use the absolute difference between the onboarding time and project creation time ($Time_{create}$) to measure the development phase of a candidate project. The following 4 features capture projects' onboarding opportunity in different perspectives.

6) First Membership Time. In some cases, although a project is created long ago by its owner, the project really starts to work until the participation of new members. We hence use the duration from the onboarding time to the first membership time ($Time_{meb0}$) to assess the lifetime of the candidate project.

7) Last Membership Time. We assume that a project has less onboarding opportunity when the project team keeps stable; and if the latest member is joined long time ago, then the project probably will not recruit new developers. Accordingly, we develop another time feature ($Time_{meb1}$) to measure the duration from the latest membership.

8) First Commit Time. For some projects, their lifecycles start from the first commit, instead of the developers' participation. Thus, we measure a project's onboarding opportunity by calculating how long a project begins to accept contributors since its first commit ($Time_{cmt0}$).

9) Last Commit Time. A project may be frequently updated by developers, although there is no newcomer joined in. We assume that this condition implies that the manpower in this project is insufficient, and this project requires more developers to share the tasks. Therefore, we adopt the last commit time ($Time_{cmt1}$) to measure the duration since the latest commit in the project.

TABLE 5: **Models for Comparison.**

Model	Description
NNLRank	The proposed list-wise ranking model.
SVMRank	A pair-wise ranking model developed by [27].
BPNet	Point-wise model, a Back-Propagation Network [47] sharing the same network structure and parameter settings as NNLRank.
SVM	Point-wise model, the support vector machine [46].
LP	The link prediction (LP) model [27].
Random	Permuting lists of candidate projects randomly.

C. BASELINE SELECTION

To assess the effectiveness of NNLRank, we compare it with a related tool LP [27] and 3 standard learning-to-rank models SVMRank [27], BPNet [28], and SVM [29]. We choose these 3 ranking models, because they are representatives of two other types of ranking model categories (i.e., pair-wise and point-wise ranking), and they are widely used for recommendation problems, as referred to in Section II. We re-implement LP adhering to the original paper details. We implement SVMRank by using the publicly available source code[3] and the SVM by invoking the WEKA[4] tool [46]. We also develop BPNet which shares the same network structure and parameter settings as our NNLRank but uses a different loss function (i.e. a common back-propagation network [47]). Finally, we develop a random model which randomly picks any project to onboard from a permuted candidate projects list. A summary of each model is given in Table 5.

D. MODEL EVALUATION

Training and Testing. In total, we have 2044 lists of candidate projects, and each list contains 257 to 2794 projects (details of the dataset described in Section V-A). To suppress the effect of outliers, we normalize 9 feature values among each list by the min-max normalization method. To verify the validity of the proposed model, we perform 5-fold cross-validation, a commonly used method [48]. At first, the whole 2044 lists of projects are randomly split into 5 sets. For each fold, one set (20% of the lists) is used for the testing data, and the rest is used for the training.

Evaluation Metrics. We use the following metrics to evaluate the performance of recommender systems. Note that, for cross-validation experiments, we measure these evaluation metrics by taking an average across 5 folds.

Mean Reciprocal Rank (MRR) - is a typical measure for evaluating the performance of a recommender system [49]. MRR equals $\frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i}$, where $rank_i$ refers to the rank position of the onboarded project for the i-th recommended list, and n is the total number of lists in testing data. MRR ranges from 0 to 1, and larger value implies that a model can successfully rank more successful onboarded projects at the top of the lists.

Mean Average Precision (MAP) - is a widely used evaluation measure [49]. For our experiment, as there is only one onboarded project for a recommended list, hence MAP is equivalent to MRR according to their definitions. However, as a recommendation rank larger than 20 tends to be meaningless in practical use, we thus use MAP@20 to cut-off the ranks larger than 20, namely $1/rank_i$=0 when $rank_i$>20. Similarly, we use MAP@10 with the cutoff 10 to assess models under a more strict condition.

Recall - we adopt Recall@10 and Recall@20 as auxiliary evaluation metrics to assess the percentage of onboarded projects [50], [51] ranked among the top- 10 and 20, respectively. A larger recall indicates that more developers can find the right projects to onboard by only reviewing at most 10/20 recommended projects.

VI. RESULTS

This section answers the Research Questions (RQs) referred to in Section III. RQ1 investigates whether the proposed model can recommend the right projects for developer onboarding effectively. RQ2 explores the relative importance of measured project features via sensitivity analysis. RQ3 compares the model execution efficiency with baseline models.

RQ1. HOW ACCURATELY CAN NNLRANK RECOMMEND ONBOARDING TO GitHub PROJECTS?

Accuracy. Table 6 presents the prediction accuracy of all models using 5-fold cross-validation, in terms of the mean and standard deviation of 5 evaluation metrics. NNLRank model achieves a MRR = 0.462, MAP@10 = 0.454, and Recall@10 = 0.611.

Compared to other evaluated models, NNLRank achieves the best performance and outperforms other models significantly. In particular, our model outperforms the second best model SVMRank by 28.69%, 29.34%, and 29.10% percentage points at MRR, MAP@10, and MAP@20 respectively. On the other hand, NNLRank shows slightly smaller Recall(@10, @20) than SVMRank, which suggests the NNLRank might not be fully trained as SVMRank to capture more developers' onboarding patterns because its 6 model parameters are manually optimized and searched within a limited scope. Besides, although SVMRank can rank slightly more onboarded projects within top-10/20 (slightly higher Recall values), their rankings are far away from the top compared with the NNLRank model, leading to substantially lower MRR and MAP. Overall, this result implies that NNLRank can enable developers to find the right projects to onboard with substantially fewer efforts.

Moreover, we perform two statistical tests between NNLRank and five baseline models, respectively, for three comprehensive evaluation metrics (MRR, MAP@10, and MAP@20) in 5-fold cross-validation. One is the Wilcoxon signed-rank test with a 5% significance level that tests the statistical difference between two models, and the other is Cliff's delta δ that quantifies the amount of difference between two models. The value of δ ranges from -1 to 1 which is divided into four effectiveness levels: negligible ($0.000 \leq |\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.330$), medium ($0.330 \leq |\delta| < 0.474$), large ($0.474 \leq |\delta| \leq 1.000$). Results in Ta-

[3]SVMRank: http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html
[4]WEKA, a data mining software in Java containing a collection of common machine learning algorithms: http://www.cs.waikato.ac.nz/ml/weka/

TABLE 6: **Prediction accuracy comparison for 6 models where an accuracy is represented by the mean ± standard deviation of 5-fold cross-validation. The statistical difference between NNLRank and a model for an evaluation metric in 5-fold cross-validation is estimated by the Wilcoxon signed-rank test at a 5% significance level, where * denotes p-value < 0.05. And we use Cliff's delta (δ) to quantify the amount of difference between two models, where L indicates a large size effect $0.474 \leq |\delta| \leq 1.000$.**

Model	MRR	MAP@10	MAP@20	Recall@10	Recall@20
NNLRank	0.462 ± 0.027	0.454 ± 0.027	0.457 ± 0.027	0.611 ± 0.037	0.659 ± 0.038
SVMRank	0.359 ± 0.022 (*L)	0.351 ± 0.023 (*L)	0.354 ± 0.022 (*L)	0.663 ± 0.034	0.716 ± 0.026
BPNet	0.018 ± 0.013 (*L)	0.013 ± 0.011 (*L)	0.014 ± 0.012 (*L)	0.030 ± 0.027	0.041 ± 0.041
SVM	0.006 ± 0.001 (*L)	0.002 ± 0.001 (*L)	0.002 ± 0.001 (*L)	0.008 ± 0.006	0.021 ± 0.011
LP	0.002 ± 0.002 (*L)	0.000 ± 0.001 (*L)	0.002 ± 0.002 (*L)	0.001 ± 0.001	0.002 ± 0.002
Random	0.008 ± 0.002 (*L)	0.004 ± 0.002 (*L)	0.005 ± 0.002 (*L)	0.012 ± 0.005	0.020 ± 0.002

ble 6 shows that the NNLRank's improvements over baseline models are significant (p-value < 0.05) and practically important with large size effect.

On the other hand, the point-wise ranking models (BPNet and SVM) can hardly work because they are strongly affected by the nature of the datasets, where only a few candidate projects are successfully onboarded by developers. This makes the dataset highly imbalanced for such point-wise ranking models. Specifically, we found that SVM usually assigns the same scores to all candidate projects, hardly capturing the features of onboarded projects, so that its recommendation performance is unsatisfactory. Although BPNet shares similar network structure and parameter settings as NNLRank, because of their distinct loss functions BPNet can generate substantially different prediction accuracy. Hence, our applied list-wise ranking loss function is one of the keys to the better performance of NNLRank. Additionally, we notice that the list-wise ranking model NNLRank outperforms the pair-wise (SVMRank) and point-wise (BPNet and SVM) ranking models, which confirms the research conclusions in other fields [52], [53].

Moreover, Table 6 shows that LP also fails to recommend projects that developers want to join and contribute many commits, although it can precisely identify developers' joining behaviors [12]. This failure is mainly caused by its inadequately used ground-truth where it assumes a joining behavior as a successful onboarding without considering the actual contribution (i.e. commits). Furthermore, this result also implies that a project joined by a developer is not necessarily to be a successful onboarding as suggested by other researchers [1], [9].

Additionally, we observe that the Random model is even better than SVM and LP because randomly permuting projects in uniform distribution has the chance to rank the onboarded one in any place, that is why Recall@10 = 0.012 and Recall@20 = 0.020. However, the probability to rank the onboarded projects within top-10 or 20 is not high, therefore MRR, MAP@10, and MAP@20 are still considerably low.

Note that, in the following experiments, we only use MAP@20 to analyze the NNLRank due to the limited space. A similar conclusion can be drawn by combining other evaluation metrics.

Impact of Development History. The independent variables for NNLRank are extracted from features of candidate projects and developers' prior experience. We assume that more developers' prior information can lead to a better model prediction. To investigate this assumption, we measure a developer's experience by the number of projects previously joined before onboarding a new project. For a testing data in 5-fold cross-validation, we divide developers' experience into 4 levels (low, medium, high, very high) based on project counts at 3 percentiles (25, 50, and 75). Table 7 shows the recommendation accuracy for developers with 4 experience levels at 5-fold cross-validation, in terms of MAP@20, where values in the corresponding parenthesis show the ranges of project counts in that level. We observe that the developers with levels in Medium and High have 2-13 previously joined projects, which implies that most of the developers prefer to contribute to multiple projects, as referred in Section V-A.

Table 7 shows that NNLRank can still perform moderately well with mean MAP@20 = 0.387 when developers have little historical information (i.e., 0-1 prior experience), and outperform the second best model SVMRank in Table 6. This is because not all independent variables are extracted from developers' prior experience. However, as expected, NNLRank works better for more experienced developers, where mean MAP@20 gradually increases from the level Low to Very High. Therefore, NNLRank works better for developers with more historical information.

TABLE 7: **Prediction accuracy (MAP@20) for developers with 4 experience levels for 5-fold cross-validation. 4 levels are divided by the 3 percentiles (20, 50, and 75) of project counts previously joined by developers at their onboarding time, where values in parenthesis show the project count ranges in that level.**

Level [Perc. Range]	Developers' Experience: Low [0-25]	Medium [25-50]	High [50-75]	Very High [75-100]
Fold-1	0.381 (0-1)	0.431 (2-4)	0.533 (5-11)	0.557 (12-260)
Fold-2	0.366 (0-1)	0.564 (2-4)	0.413 (5-13)	0.491 (14-298)
Fold-3	0.405 (0-1)	0.358 (2-3)	0.469 (4-11)	0.526 (12-273)
Fold-4	0.324 (0-1)	0.351 (2-3)	0.485 (4-10)	0.573 (11-187)
Fold-5	0.461 (0-1)	0.568 (2-3)	0.540 (4-11)	0.467 (12-225)
Mean MAP@20	0.387	0.454	0.488	0.523

Result 1: NNLRank can make effective project onboarding recommendations for developers, outperforming the other investigated models substantially. NNLRank works better for developers with more historical information, and it still outperforms other investigated models even with little developers' history.

TABLE 8: **Prediction accuracy (MAP@20) of NNLRank inputted with different feature combinations for sensitivity analysis. Each combination removes one feature, and the 'Order' column indicates the rank of features' relative importance.**

Removed Feature	MAP@20	Order	Removed Feature	MAP@20	Order
$Time_{create}$	0.351	1	Tie_{owner}	0.444	6
$Time_{meb0}$	0.422	2	$User_{company}$	0.452	7
$Time_{cmt0}$	0.431	3	$Tech_{lang}$	0.453	8
$Time_{meb1}$	0.436	4	$Time_{cmt1}$	0.454	9
$Info_{cmt}$	0.438	5	-	-	-

TABLE 9: **Prediction accuracy of NNLRank incorporating project characteristics. Without such feature MAP@20 = 0.457.**

Model Setting	MAP@20
NNLRank+Des_{sum}	0.405
NNLRank+$Read_{sum}$	0.413
NNLRank+$ReadDes_{sum}$	0.413
NNLRank+Des_{max}	0.399
NNLRank+$Read_{max}$	0.402
NNLRank+$ReadDes_{max}$	0.389

RQ2. WHAT ARE THE MOST IMPORTANT FEATURES THAT INFLUENCE ONBOARDING RECOMMENDATION?

Here we investigate the relative importance of 9 measured features using sensitivity analysis as illustrated in Table 8. Specifically, we remove one feature at a time and record the model prediction accuracy with the other 8 features, where a more influential feature is the one with more negative impact on MAP@20 (i.e. less MAP), and their relative importance order is also listed in the Table.

Table 8 shows that all of features have positive effects on the model; the time-related features have higher importance, followed by project commits ($Info_{cmt}$), social tie (Tie_{owner}), and language matching ($Tech_{lang}$). These results imply that the project has more onboarding opportunities at its early phase; developers prefer to work with a project owner with whom they have frequently collaborated; if the project matches developers' prior experience in programming language, and if the project has many commits and company colleagues, developers can successfully onboard the project with higher probability.

One may expect that developer having expertise in one project may want to contribute to functionally similar projects. To check this hypothesis, we first identify functionally similar projects. In particular, we implement a Latent Semantic Indexing (LSI) [54] based similarity model following Mcmillan et al. [13], which leverages project description and readme to compute the functional similarity score between two GitHub projects. We further designed 6 features as listed in Table 9, where Des_{sum}, $Read_{sum}$, and $ReadDes_{sum}$ are the sum of similarity values between a candidate project and a developer's prior joined projects using the projects' description, readme, or both; in contrast, Des_{max}, $Read_{max}$, and $ReadDes_{max}$ use the maximum similarity value instead of the sum.

Surprisingly, we find that adding these project semantic features to NNLRank have negative impacts on model performance (see Table 9). This implies that developers do not always prefer to work on functionally similar projects. For instance, consider two functionally similar projects (e.g., media player) written in Java and C/C++. For a developer, whose expertise is in only Java will choose the project written in Java. Thus, projects belong to the similar functional domain may not always influence developers' onboarding decisions. To this end, we ignore these features from our final model architecture.

> **Result 2:** All 9 measured features have positive effects on NNLRank, where time related features have higher importance because projects provide more onboarding opportunity at early lifecycle.

RQ3. WHAT IS THE RUN-TIME OVERHEAD OF NNLRANK?

Table 10 illustrates the execution time of all models, running on a duo core computer (2.4 GHz) with 4 GB memories, where the execution time indicates the time of model training and testing for 5-fold cross-validation. Table 10 shows that NNLRank and the second best effective model, SVMRank (details in RQ1) respectively cost 20.435s and 250.291s to execute. NNLRank's substantially smaller execution time indicates that it can efficiently work because other models cannot make effective recommendations as shown in Table 6. NNLRank works fast because it processes a list of candidate projects simultaneously, which can dramatically save the time of model building.

Besides, we notice that although NNLRank and BPNet share the same network structure and parameter setting, their difference in loss function leads to some different execution time. The small execution time of BPNet (10.283s) over NNLRank implies that the list-wise ranking loss function in NNLRank has a higher computation complexity. Furthermore, the efficiency of NNLRank is comparable to the LP model (21.049s) and substantially faster than SVM (497.658s).

Efficiency for NNLRank with Different Settings. We further investigate two important parts of NNLRank which are instrumental to the efficiency: the learning rate decay method, and the capability of processing projects simultaneously. In Table 11, we observe that when removing the learning rate decay method from the NNLRank, the execution time increases from 20.435s to 32.925 due to the largely increased optimization iteration numbers; and its MAP@20 largely drops for the incompatible parameters, which implies that the learning rate decay method can determine optimal parameter selection. We also observe that when the NNLRank processes each project separately with or without the learning rate decay method, its execution time is significantly increased, because this model setting works like the point-wise ranking model (i.e. SVM).

TABLE 10: **Execution time comparison of 6 models, where an execution time indicates the time of model training and testing for 5-fold cross-validation.**

Model	Time	Model	Time
BPNet	10.283s	SVMRank	250.291s
NNLRank	20.435s	SVM	497.658s
LP	21.049s	-	-

TABLE 11: **Comparison of NNLRank with different settings: with or without learning rate decay method; process a list of candidate projects simultaneously or separately. Results in execution time, average model optimization iterations, and MAP@20 are compared.**

Setting	Time	Avg. Iterations	MAP@20
NNLRank	20.435s	52.60	0.457
NNLRank - Decay	32.925s	326.80	0.001
NNLRank - Simu.	157.915s	58.40	0.447
NNLRank - Decay - Simu.	648.636s	203.80	0.001

Result 3: NNLRank can be quickly trained and used for project recommendation, due to its ability to process a list of projects simultaneously, and the use of learning rate decay method.

VII. DISCUSSION

This section discusses how the model parameters affect its performance. We explain the results for MAP@20 and the similar conclusions can be drawn for other evaluation metrics. Specifically, we set 6 parameters in NNLRank as default as shown in Table 12, and adjust one at a time as follows.

Network Structure. We used different model structures (i.e. number of hidden layers, and the number of nodes in each layer) for onboarding task. Figure 2(a) shows the impact of different hidden layers. NNLRank performs best at 2 hidden layers, thus we choose this for our final model. Figure 2(b) shows the impact of the node count in hidden layers. We see that the network performs better at 5 nodes and performance remain similar with the increase in the number of nodes. As an increase in nodes raises the computation complexity, the network structure with 5 nodes is optimal for our model.

Activation Function. Activation function plays an important role in the neural network performance. Thus, we further investigated five commonly used activation functions. In Table 13, we see that Arctan is the best choice with MAP@20 = 0.457. The Bent, SoftPlus, linear activation functions fail to work, might be because of the incompatible parameters.

TABLE 12: **Default Parameter Setting**

No.	Parameter	Default Value
1	Number of Hidden Layers	2.00
2	Number of Nodes in Hidden Layers	5.00
3	Regularization Coefficient (λ)	1.00
4	Learning Rate (μ)	0.01
5	The Initial Step for Learning Rate Decay (t_{after})	0.00
6	Convergence Error (ε)	0.10

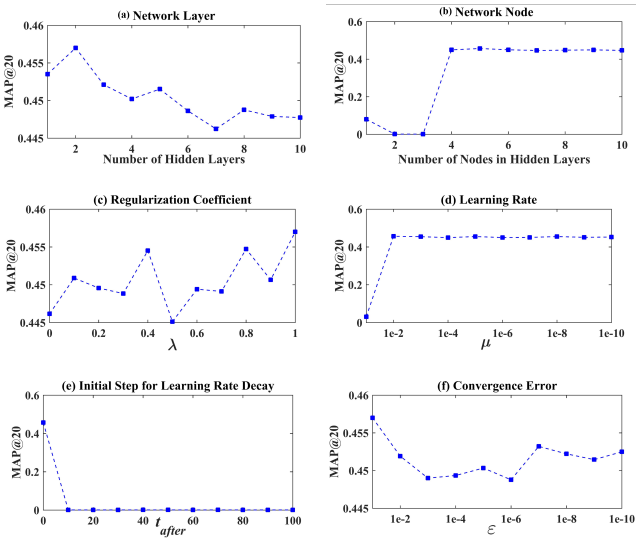


FIGURE 2: **Prediction accuracy (MAP@20) of NNLRank with different configurations, adjusting the number of hidden layers, the number of nodes in hidden layers, the regularization coefficient (λ), the learning rate (μ), the initial step for learning rate decay (t_{after}), and the convergence error (ε).**

Thus, Arctan is the best choice for NNLRank.

TABLE 13: **Prediction accuracy of NNLRank with 5 different activation functions.**

No.	Activation Function	Equation	MAP@20
1	ArcTan	$y = atan(x)$	0.457
2	Sigmoid	$y = 1/(1+e^{-x})$	0.446
3	Bent	$y = ((x^2+1)^{1/2})/2 + x$	0.000
4	SoftPlus	$y = log(1+e^x)$	0.000
5	Linear	$y = x$	0.000

Network Parameters. We tune four hyper-parameters - regularization coefficient (λ), learning rate (μ), the initial iteration for learning rate decay (t_{after}), and convergence error (ε) and Figures 2(c-f) show the network performance for different values respectively. We see that the model achieves the best performance when λ=1, which implies that a lower complexity of weights and biases is beneficial to the model. Moreover, we find that updating the network with μ=0.01 is advantageous, where the smaller values for μ considerably decrease the performance, and higher values have negligible improvements. The learning rate decay strategy is beneficial to the model at the beginning (i.e., t_{after}=0), as the prediction accuracy slides down when t_{after} increases. Finally, we find that ε=0.1 is suitable for NNLRank because larger values not only decrease the model accuracy but also increase the iteration number of the model training, which is also a reason why NNLRank works efficiently.

Weights Initialization. We further analyze how the randomly initialized weights affect the recommendation accuracy of NNLRank. Figure 3 shows the performance of NNLRank for 100 consecutive simulations where each time

we randomly initialize model weights. We see that the model performance remain similar for different random initializations. This result shows that the simulations are stable, where MAP@20 ranges from 0.447 to 0.463 with a standard deviation of 0.003 which indicates the randomness in weights initialization has little impact on the model performance. Thus, we choose to initialize weights randomly.

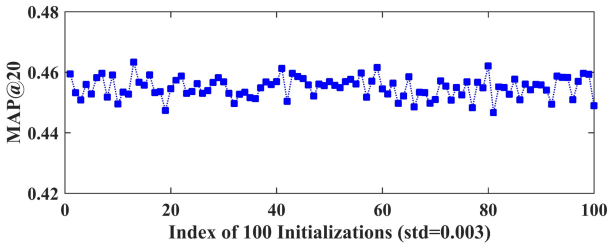


FIGURE 3: **Prediction accuracy (MAP@20) of 100 consecutive weights initializations for the neural network in NNLRank, with a standard deviation (std) value.**

VIII. IMPLICATION

Implication to OSS Ecosystem: To utilize the effectiveness of NNLRank for developer onboarding, an ecosystem such as GitHub may provide an option for onboarding recommendation. Faster run-time enables easy adoption of NNLRank to any existing ecosystem. Moreover, a pre-trained NNLRank can be used to recommend projects to developers.

Helping Newbies: Experiment result shows that the NNLRank remains effective for developers with little historical information, i.e. newbies, which implies that the NNLRank can avoid the cold start problem to some extent. However, the model effectiveness can be largely improved for developers with sufficient histories. Therefore, to improve the utility of NNLRank in practical usage, developers can import their prior experience from other platforms, such as SourceForge; or they can simply fill in their prior project experience and preference, and input to the NNLRank to get a better recommendation.

Helping Expert Developer: As NNLRank works better for developers with more histories, the project recommendation is more likely to be accepted by an expert. This advantage can significantly save developers' efforts for project searching, understanding, and filtering.

IX. THREATS TO VALIDITY

To extract the project time-related features, we need to know developers' onboarding time. However, such time is hard to measure because a developer may decide to onboard a project at any time, and an inaccurate measurement may compromise the model validity. To minimize this threat, we measured developers' onboarding decisions at the time of their first commits following Casalnuovo et al. [1] and Hahn et al. [42].

In this study, we measured 9 features, i.e. the independent variables, to capture developers' onboarding pattern. However, it has been observed that many other features also affecting developers' project selection [42]. Due to the nature of our data from GitHub, we could not extract these features such as the number of downloads, mailing list, and etc. We plan to explore more viable features in the future.

Due to the enormous size of the downloaded data and our limited computational resources, we studied 2044 successful onboarding decisions within 2 years made by GitHub developers, as referred in Section V-A. We plan to work on a larger dataset with more recent information in near future. Additionally, our model is verified by a 5-fold cross-validation by randomly partitioned data into folds. The inherent limitation of such measure is that the model may be trained on developers' future information. However, as developers' onboarding decisions are usually independent of each other, the impact of this threat is negligible. Moreover, we use the same experimental setup for all models, thus comparison results are not affected by this threat. Furthermore, we re-implement the previous tool, LP following the original paper [12]. However, it may still have some biases threating the validity of our conclusion, but we went through a rigorous code review to minimize this threat.

X. CONCLUSION

Though onboarding plays an important role in the evolutionary progress of open source ecosystems, developers often face enormous challenges to find suitable projects. In this paper, we propose a learning to rank based model NNLRank which helps developers by recommending relevant projects for onboarding. NNLRank leverages project features and developers' experience to predict appropriate projects. We evaluated NNLRank with 2044 successful onboarding decisions from GitHub and compared it with three standard learning-to-rank models and a prior onboarding tool. Experimenting with 2044 successful onboarding decisions, we confirmed the effectiveness and efficiency of NNLRank model where it substantially outperforms previous models for onboarding task.

ACKNOWLEDGMENT

The work described in this paper was partially supported by the Fundamental Research Funds for the Central Universities of China (Grant No. 106112017CDJXSYY002), the National Natural Science Foundation of China (Grant no. 61772093), and Chongqing Research Program of Basic Science & Frontier Technology (Grant No. cstc2017jcyjB0305).

REFERENCES

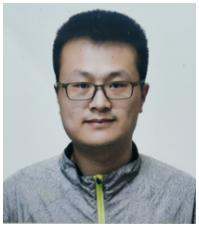
[1] C. Casalnuovo, B. Vasilescu, P. Devanbu, and V. Filkov, "Developer onboarding in github: the role of prior social links and language experience," in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015, pp. 817–828.

[2] B. Vasilescu, V. Filkov, and A. Serebrenik, "Perceptions of diversity on git hub: A user survey," in Cooperative and Human Aspects of Software Engineering (CHASE), 2015 IEEE/ACM 8th International Workshop on. IEEE, 2015, pp. 50–56.

[3] K. Crowston and J. Howison, "The social structure of free and open source software development," First Monday, vol. 10, no. 2, 2005.

[4] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: cross-project code reuse in github," in Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, 2017, pp. 291–301.

[5] K. Crowston, Q. Li, K. Wei, U. Y. Eseryel, and J. Howison, "Self-organization of teams for free/libre open source software development," Information and software technology, vol. 49, no. 6, pp. 564–575, 2007.
[6] R. Guimera, B. Uzzi, J. Spiro, and L. A. N. Amaral, "Team assembly mechanisms determine collaboration network structure and team performance," Science, vol. 308, no. 5722, pp. 697–702, 2005.
[7] I. Steinmacher, M. A. G. Silva, M. A. Gerosa, and D. F. Redmiles, "A systematic literature review on the barriers faced by newcomers to open source software projects," Information and Software Technology, vol. 59, pp. 67–85, 2015.
[8] I. Steinmacher, T. Conte, M. A. Gerosa, and D. Redmiles, "Social barriers faced by newcomers placing their first contribution in open source software projects," in Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing. ACM, 2015, pp. 1379–1392.
[9] M. Zhou and A. Mockus, "What make long term contributors: Willingness and opportunity in oss community," in Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012, pp. 518–528.
[10] L. Torgamus, "How can I find a good open source project to join?" https://arstechnica.com/information-technology/2012/03/ask-stack-how-can-i-find-a-good-open-source-project-to-join/, 2012, [Online; accessed 26-April-2018].
[11] ——, "How can I find a good open source project to join?" https://eve.community/t/contributing-to-projects/6798, [Online; accessed 26-April-2016].
[12] T. Matek and S. T. Zebec, "Github open source project recommendation system," arXiv preprint arXiv:1602.02594, 2016.
[13] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012, pp. 364–374.
[14] K. R. Lakhani and E. Von Hippel, "How open source software works: "free" user-to-user assistance," Research policy, vol. 32, no. 6, pp. 923–943, 2003.
[15] Y. Ye and K. Kishida, "Toward an understanding of the motivation open source software developers," in Proceedings of the 25th international conference on software engineering. IEEE Computer Society, 2003, pp. 419–429.
[16] L. Torgamus, "https://opensource.guide/how-to-contribute/," https://opensource.guide/how-to-contribute/, 2018, [Online; accessed 26-April-2018].
[17] CodeTriage, "CodeTriage.com," https://www.codetriage.com/, 2018, [Online; accessed 26-April-2018].
[18] OpenHatch, "OpenHatch.org," https://openhatch.org/, 2018, [Online; accessed 26-April-2018].
[19] I. Steinmacher, M. A. Gerosa, and D. Redmiles, "Attracting, onboarding, and retaining newcomer developers in open source software projects," in Workshop on Global Software Development in a CSCW Perspective, 2014.
[20] I. Steinmacher, I. S. Wiese, T. Conte, M. A. Gerosa, and D. Redmiles, "The hard life of open source software project newcomers," in Proceedings of the 7th international workshop on cooperative and human aspects of software engineering. ACM, 2014, pp. 72–78.
[21] I. Steinmacher, A. P. Chaves, T. U. Conte, and M. A. Gerosa, "Preliminary empirical identification of barriers faced by newcomers to open source software projects," in Software Engineering (SBES), 2014 Brazilian Symposium on. IEEE, 2014, pp. 51–60.
[22] GitHub, "Discover repositories," https://github.com/dashboard/discover, 2018, [Online; accessed 26-April-2018; Login Required].
[23] Y. Malheiros, A. Moraes, C. Trindade, and S. Meira, "A source code recommender system to support newcomers," in Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual. IEEE, 2012, pp. 19–24.
[24] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Learning from project history: a case study for software development," in Proceedings of the 2004 ACM conference on Computer supported cooperative work. ACM, 2004, pp. 82–91.
[25] ——, "Hipikat: A project memory for software development," IEEE Transactions on Software Engineering, vol. 31, no. 6, pp. 446–465, 2005.
[26] H. B. Demuth, M. H. Beale, O. De Jess, and M. T. Hagan, Neural network design. Martin Hagan, 2014.
[27] T. Joachims, "Training linear svms in linear time," in Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2006, pp. 217–226.
[28] R. Hecht-Nielsen et al., "Theory of the backpropagation neural network." Neural Networks, vol. 1, no. Supplement-1, pp. 445–448, 1988.
[29] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," Neural processing letters, vol. 9, no. 3, pp. 293–300, 1999.
[30] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," IEEE software, vol. 27, no. 4, pp. 80–86, 2010.
[31] M. Gasparic and A. Janes, "What recommendation systems for software engineering recommend: A systematic literature review," Journal of Systems and Software, vol. 113, pp. 101–113, 2016.
[32] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching documents with examples: A corpus mining approach," ACM Transactions on Information Systems (TOIS), vol. 31, no. 1, p. 1, 2013.
[33] G. Robles and J. M. Gonzalez-Barahona, "Contributor turnover in libre software projects," in IFIP International Conference on Open Source Systems. Springer, 2006, pp. 273–286.
[34] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. Van Deursen, "Communication in open source software development mailing lists," in Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on. IEEE, 2013, pp. 277–286.
[35] H. Ma, D. Zhou, C. Liu, M. R. Lyu, and I. King, "Recommender systems with social regularization," in Proceedings of the fourth ACM international conference on Web search and data mining. ACM, 2011, pp. 287–296.
[36] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," IEEE transactions on knowledge and data engineering, vol. 17, no. 6, pp. 734–749, 2005.
[37] T.-Y. Liu et al., "Learning to rank for information retrieval," Foundations and Trends® in Information Retrieval, vol. 3, no. 3, pp. 225–331, 2009.
[38] T. Qin, X.-D. Zhang, M.-F. Tsai, D.-S. Wang, T.-Y. Liu, and H. Li, "Query-level loss functions for information retrieval," Information Processing & Management, vol. 44, no. 2, pp. 838–855, 2008.
[39] F. Xia, T.-Y. Liu, J. Wang, W. Zhang, and H. Li, "Listwise approach to learning to rank: theory and algorithm," in Proceedings of the 25th international conference on Machine learning. ACM, 2008, pp. 1192–1199.
[40] R. Herbrich, T. Graepel, and K. Obermayer, "Support vector learning for ordinal regression," 1999.
[41] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," Journal of machine learning research, vol. 4, no. Nov, pp. 933–969, 2003.
[42] J. Hahn, J. Y. Moon, and C. Zhang, "Emergence of new project teams from open source software developer networks: Impact of prior collaboration ties," Information Systems Research, vol. 19, no. 3, pp. 369–391, 2008.
[43] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks." in Aistats, vol. 9, 2010, pp. 249–256.
[44] J. Hu, J. Lu, and Y.-P. Tan, "Discriminative deep metric learning for face verification in the wild," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 1875–1882.
[45] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in Proceedings of the 11th working conference on mining software repositories. ACM, 2014, pp. 92–101.
[46] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," ACM SIGKDD explorations newsletter, vol. 11, no. 1, pp. 10–18, 2009.
[47] S. Haykin and N. Network, "A comprehensive foundation," Neural Networks, vol. 2, no. 2004, p. 41, 2004.
[48] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation," in Encyclopedia of database systems. Springer, 2009, pp. 532–538.
[49] K. Balog, T. Bogers, L. Azzopardi, M. De Rijke, and A. Van Den Bosch, "Broad expertise retrieval in sparse data environments," in Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 2007, pp. 551–558.
[50] R. M. Bolle, J. H. Connell, S. Pankanti, N. K. Ratha, and A. W. Senior, "The relation between the roc curve and the cmc," in Automatic Identification Advanced Technologies, 2005. Fourth IEEE Workshop on. IEEE, 2005, pp. 15–20.
[51] S. M. Assari, H. Idrees, and M. Shah, "Human re-identification in crowd videos using personal, social and environmental constraints," in European Conference on Computer Vision. Springer, 2016, pp. 119–136.
[52] A. Trotman, "Learning to rank," Information Retrieval, vol. 8, no. 3, pp. 359–381, 2005.
[53] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, "Learning to rank: from pairwise approach to listwise approach," in Proceedings of the 24th international conference on Machine learning. ACM, 2007, pp. 129–136.
[54] T. K. Landauer and S. T. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge." Psychological review, vol. 104, no. 2, p. 211, 1997.



CHAO LIU received the M.S. degree in Software Engineering from Chongqing University, Chongqing, PR China, in 2014 and B.S. degree in 2011. He is currently pursuing the Ph.D. degree in Software Engineering at Chongqing University, Chongqing, China. His research interest includes the Information Retrieval, Machine Learning, Deep Learning, Data Mining, and Program Analysis.

DAN YANG received the Ph.D. degree from Chongqing University, PR China in 1997, where he also received the M.S. degree in Applied Mathematics in 1985 and B.S. degree in 1982. He is a Professor in School of Big Data & Software Engineering at Chongqing University. His current research interests include Data Mining of Software Engineering, Topic Modeling, Image Semantic Analysis, and Video Analysis.

XIAOHONG ZHANG received the Ph.D. degree in Computer Software and Theory from Chongqing University, PR China in 2006, where he also received the M.S. degree in Applied Mathematics. He is a Professor in School of Big Data & Software Engineering at Chongqing University. His current research interests include Data Mining of Software Engineering, Topic Modeling, Image Semantic Analysis, and Video Analysis.

BAISHAKHI RAY is an Assistant Professor in the Department of Computer Science, Columbia University, NY, USA. She has received her Ph.D. degree in Electrical & Computer Engineering from the University of Texas, Austin and masters degree in Computer Science from the University of Colorado, Boulder. Baishakhi's research interest is in Software Engineering with the intent to improve software reliability and security. Baishakhi has received Best Paper awards at FSE 2017, MSR 2017, IEEE Symposium on Security and Privacy (Oakland), 2014. Her research has also been published in CACM Research Highlights and has been widely covered in trade media.

MD MASUDUR RAHMAN received the B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology, Bangladesh, in 2013. He was a lecturer in Computer Science and Engineering department at BRAC University, Bangladesh. He is currently pursuing the Ph.D. degree in Computer Science at University of Virginia, VA, USA. His current research interests include Information Retrieval, Natural Language Processing, Machine Learning, Code Search, Software Engineering, and Big Code.