Empirical Software Engineering (2020) 25:859–889
https://doi.org/10.1007/s10664-019-09755-0

Developer recommendation for Topcoder through a meta-learning based policy model

Zhenyu Zhang[1,2] · Hailong Sun[1,2] · Hongyu Zhang[3]

Published online: 5 September 2019

Abstract
Crowdsourcing Software Development (CSD) has emerged as a new software development paradigm. Topcoder is now the largest competition-based CSD platform. Many organizations use Topcoder to outsource their software tasks to crowd developers in the form of open challenges. To facilitate timely completion of the crowdsourced tasks, it is important to find right developers who are more likely to win a challenge. Recently, many developer recommendation methods for CSD platforms have been proposed. However, these methods often make unrealistic assumptions about developer status or application scenarios. For example, they consider only skillful developers or only developers registered with the challenges. In this paper, we propose a meta-learning based policy model, which firstly filters out those developers who are unlikely to participate in or submit to a given challenge and then recommend the top k developers with the highest possibility of winning the challenge. We have collected Topcoder data between 2009 and 2018 to evaluate the proposed approach. The results show that our approach can successfully identify developers for posted challenges regardless of the current registration status of the developers. In particular, our approach works well in recommending new winners. The accuracy for top-5 recommendation ranges from 30.1% to 91.1%, which significantly outperforms the results achieved by the related work.

Keywords Topcoder · Developer recommendation · Meta-learning · Crowdsourcing software development

Communicated by: Kelly Blincoe

✉ Hailong Sun
sunhl@buaa.edu.cn

Zhenyu Zhang
zhangzhenyu13@outlook.com

Hongyu Zhang
hongyu.zhang@newcastle.edu.au

1 SKLSDE Lab, School of Computer Science and Engineering, Beihang University, Beijing, 100191, China

2 Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing, 100191, China

3 The University of Newcastle, Callaghan, Australia

1 Introduction

Crowdsourcing Software Development (CSD), which outsources software tasks to the crowd developers, is an emerging software development paradigm (Begel et al. 2012; Dubey et al. 2017; Saremi and Yang 2015; Saremi et al. 2017). Many organizations like Amazon, Google, NASA, and Microsoft have utilized the services of the CSD platforms to solicit contributions from talented developers across the globe (Hasteer et al. 2016).

The general procedure of CSD usually involves three types of roles (Stol and Fitzgerald 2014), namely customers, workers, and platforms. Customers can post tasks in a platform. Workers are the crowd developers who conduct the tasks outsourced by the customers and submit their contributions via the platform. The platform acts as a marketplace for the workers and customers, and maintains all the issues and artifacts created by them. To encourage the community to participate in crowdsourcing software development, many CSD platforms, such as Topcoder, take the form of open contests, where each task is treated as a *challenge*. For a posted challenge, developers can register with it and submit their contributions to it. The winner (usually 1 or 2 developers) will get paid while the inexperienced developers will get credits.

There is a wide range of research on CSD (Mao et al. 2017; Stol and Fitzgerald 2014; Hasteer et al. 2016; Zanatta et al. 2018; Dubey et al. 2017; Abhinav et al. 2017; Cui et al. 2017; Saremi and Yang 2015), but still a lot of problems in CSD remain unsolved. One of the most important issues are the recommendation of reliable developers because most crowd developers do not submit any work after registration, which can be harmful for time-critical challenges. For example, according to our data collected from Topcoder, about 85% of developers have ever registered with a challenge, but only around 23% of the registrants have submitted their works. The high quitting rate is harmful to crowdsourcing software development.

In order to improve the efficiency of CSD, many researchers proposed models to recommend reliable developers to a crowdsourced task so that the task can be finished on time with quality. For example, some researchers treated the recommendation problem as a multi-class classification problem (Mao et al. 2015; Fu et al. 2017). They utilized clustering and classification algorithms to recommend developers for the Topcoder challenges. Yang et al. (2016) proposed a recommender system that can help crowd developers make decisions in participating in a challenge. However, there are three major problems in current recommendation models:

- Unrealistic assumptions. The existing methods for developer recommendation in CSD make several important assumptions about developer status or application scenarios. For example, the methods proposed in Mao et al. (2015) and Fu et al. (2017) only concern skillful developers (i.e. those who have won the challenges for 5 times at least). However, our statistics show that developers who won over 5 challenges take up no more than 10% of all winners and many winners only have 1 or 2 winning records. The work proposed in Yang et al. (2016) predicts if a developer has a chance to win a challenge once the registration status of the developer is known (i.e. whether a developer has registered with the challenge or not). However, for many developers such status is not known beforehand. The existing methods make the above assumptions because they formulate the developer recommendation problem as a multi-class classification problem, therefore they need to fix the number of classes (developers) to a relatively small set, which is unrealistic in practice.

- No support for new winners. The current methods predict winners through multi-class classification, where labels are a fixed set of developers who have winning histories. Therefore, the current methods are unable to predict a potential winner who has never won before. This can be viewed as a form of the cold-start problem.
- Low accuracy. Because of the above two problems, the recommendation accuracy of the existing methods is still not satisfactory (e.g., the top-5 recommendation accuracy is lower than 40% on most of our datasets) if we consider all the challenges (without filtering out developers with few winning records).

In our work, we build a policy model to address the problems mentioned above. We model the developer recommendation process using the policy model, which consists of a sequence of procedures for predicting registration, submission, and winning status of a developer, respectively. Only the developers who are likely to register and submit to a challenge are used in winner prediction, thus relieving the necessity of assuming developer status. There are many factors that we need to consider in the design of the policy model, such as different machine learning algorithms and different threshold settings. To achieve an optimal solution, we adopt the meta-learning paradigm (Brazdil et al. 2008; Metalearning 2009; Rice 1976) to automatically select proper machine learning algorithms and tune threshold parameters. In our meta-learning approach, the space of meta-features (algorithms and parameters) is searched and the optimal ones that can achieve the best overall prediction performance are selected. The resulting policy model with the optimal meta-features is used for developer recommendation.

We have experimented with the proposed approach using the real-world data collected from Topcoder. We train our models on 11 Topcoder datasets and test the models using the recently posted challenges. The results show that the proposed approach outperforms the existing methods and can recommend developers who have only a few or even no winning records. Furthermore, some types of challenges are newly proposed by CSD platforms and contain only a small number of historical records. Our source code and experimental data are available in GitHub.[1]

Our work can help crowdsourced projects find suitable developers and facilitate timely completion of the project. Although our evaluation is performed on Topcoder data only, the general principles of the proposed approach is applicable to other CSD platforms as well. The major contributions of the paper are as follows:

- We propose a meta-learning based policy model for recommending developers in CSD. Our model does not make assumptions about developer status and is therefore more realistic in practice. Furthermore, our approach can support new developers and challenge types.
- We have conducted extensive experiments on 11 major Topcoder datasets. The results confirm the effectiveness of the proposed approach.

The rest of the paper is organized as follows: Section 2 describes the background and the Topcoder dataset. Section 3 describes the proposed developer recommendation method for CSD. Section 4 evaluates the effectiveness of our recommender system and analyzes the results of the experiments. Section 5 discusses the model capacity in supporting new winners and challenge types. We show the threats to validity in Section 6, introduce related work in Section 7, and conclude the paper in Section 8.

[1] https://github.com/zhangzhenyu13/CSDMetalearningRS

2 Background

2.1 Crowdsourcing Software Development

Crowdsourcing Software Development (CSD) has been increasingly adopted in recent years. CSD outsources software tasks to crowd developers and has the advantages of low-cost, short time-to-market, and open innovation (Hasteer et al. 2016). Many companies such as Google and Microsoft have successfully used CSD platforms to develop software components. CSD is also an active topic in recent software engineering research (Begel et al. 2012; Saremi et al. 2017; Khanfor et al. 2017; Abhinav et al. 2017; Zanatta et al. 2018).

There are many popular CSD platforms such as Topcoder,[2] Freelancer,[3] Upwork,[4] and Kaggle.[5] They all adopt an Open Call form to attract developers to contribute to the posted tasks. To facilitate developer participation, many CSD platforms take the form of open contests, where each task is treated as a *challenge* and developers compete in the challenge. Take Topcoder as an example, the development process consists of challenge posting, developer registration, work submission, and then work reviewing. Finally the winners are selected and awarded.

2.2 Developer Recommendation for CSD

In competition-based CSD platforms, developers need to consider whether or not to register and submit their work in order to win the competition. According to our study of Topcoder, nearly 2/3 of posted challenges fail to complete due to zero submission. Among the completed challenges, only 23% of developers have ever submitted their work. Therefore, it is beneficial to build a model to identify the potential winners for customers of the posted challenges (i.e., challenge organizers) and then recommend these developers to the customers. Developer recommendation is especially helpful for the challenges that receive few submissions or even few registrations. The challenge organizers can proactively contact the recommended developers regarding the crowdsourced work.

Recently, some developer recommendation methods for CSD have been proposed. Fu et al. (2017) proposed a clustering based collaborative filtering classification model (CBC), which formulates the winner prediction problem as a multi-label classification problem. Their best experimental results are achieved when Naive Bayes classifier is used. They also proposed a competition network, which further helps to improve the recommendation accuracy slightly. Mao et al. (2015) proposed a recommender system called CrowdRex, which extracts developers' history data and challenge data as input for their model. Their best results are achieved when using decision tree as the classifier. Both CBC and CrowdRex only take into consideration skillful developers who have at least 5 winning records. However, most of the developers only win no more than 2 times. Therefore, excluding the developers with fewer than 5 winning records is unrealistic in practice. Yang et al. (2016) leveraged several influential factors to build a dynamic crowd worker decision support model (DCW-DS), which can predict a developer's role (registrant, submitter, or winner) for a given challenge. They obtained the best results when the Random Forest classifier is

[2] https://www.topcoder.com/

[3] https://www.freelancer.ca/

[4] https://www.upwork.com/

[5] https://www.kaggle.com/competitions/

used. However, the performance of their method is not satisfactory when no registration or submission status is observed.

2.3 Topcoder Dataset

We use Topcoder as an exemplar CSD platform to describe our approach throughout the paper. Topcoder is now the world's largest service provider of competition-based CSD. There are many types of tasks in Topcoder. For example, "Test Suites" focuses on testing correctness of a posted challenge in this category, "Assembly" focuses on integrating components of a software project, "Bug Hunt" aims at finding bugs, etc. In the rest of the paper, the term *type* refers to the category of a task and the term *challenge* refers to a concrete task instance of a type. We divide all challenges according to their types into different *datasets*. Each dataset contains all challenges of that type of tasks and the corresponding developers that participate in those challenges.

Figure 1 gives an example of a Topcoder challenge. The "Subtrack" field contains a list of types. The list panel in the bottom contains the challenges that developers can choose to participate (register and submit). Figure 1 also shows an example task ("Delta Migration from Postgres to Informix"), which contains the task description, required techniques, important dates, prizes, current registrants, submissions, etc. In this example, there are 0 submission and 28 registrants. In fact, our statistics show that there are no more than 100 registrants for over 90% of the posted challenges. For each challenge, many developers fail to register or submit. And among the registrants, half of them quit the challenge. In our work, we facilitate timely completion of the tasks by recommending suitable developers.

3 Meta-Learning Based Policy Model for Developer Recommendation

3.1 Overall Design

The nature of challenge-based CSD consists of three phases: registration, submission, and winning. Taking these three phases into consideration, we construct a policy model for developer recommendation. Such a policy model reflects the fact that developer recommendation in competition-based CSD is a sequence of registration, submission, and winning: developers cannot make any submission if they do not register with the challenge and they cannot win if they make no submission.

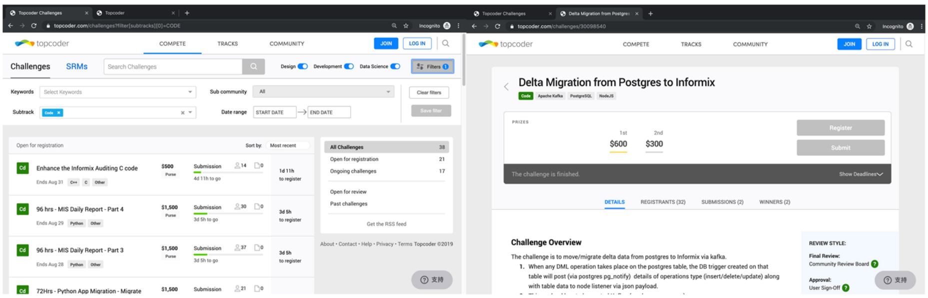


Fig. 1 An example of a Topcoder challenge

Our policy model contains three machine-learning based predictors including the registration, submission, and winning predictors. Each predictor can output a probability value for a developer and rank all developers according to this value. The two thresholds top_R and top_S that range from 0.0 (0%) to 1 (100%) are for determining the top_R and top_S of developers who can succeed in registration and submission, respectively. The workflow of the three predictors is shown in Fig. 2. The variable $P(Win)$ indicates the probability of a developer being a winner. For example, if the rank of a developer given by registration predictor is not within the top_R of all developers of a posted challenge, the developer is considered inactive in registration and the corresponding winning probability is set to 0. Otherwise, the model will predict the developer's submission and winning status in the follow-up steps.

Our overall objective is to find proper machine learning algorithms and threshold parameters that can maximize the prediction performance of the policy model. The optimal algorithms and parameters (top_R and top_S) are obtained through meta-learning (Section 3.4). Improper threshold parameters could affect the accuracy of the model adversely. For example, a very small value of top_R may filter away too many developers, which is harmful to recommendation. While a large top_R may include nearly all the developers, which is also harmful to recommendation. The final winning probability value given by winning predictor is used to rank the developers. The top k developers in the list are recommended to the customers (the challenge organizers).

The overall structure of our approach is illustrated in Fig. 2. The recommender system contains the following three components:

- Data extractor, which extracts features from the challenge and developer data and constructs the input data for base predictors.
- Base predictors, which predict the probability of a developer registering with, submitting to, and winning a challenge, respectively. Each predictor consists of three machine learning algorithms including ExtraTrees, XGBoost, and Neural Network.

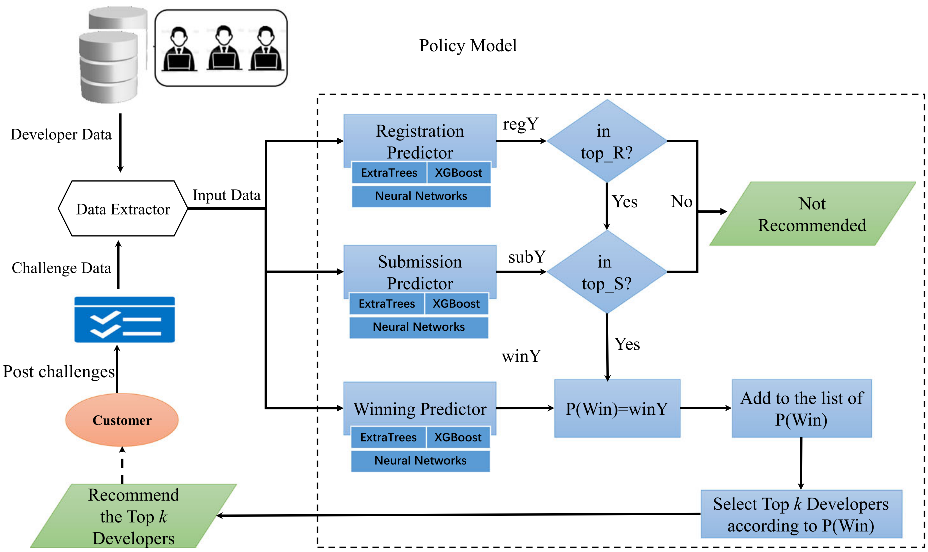


Fig. 2 Our meta-learning based recommender system

– Policy model, which consists of three base predictors and uses the previously learned meta features as the policy to filter out those developers that are impossible to win. Finally, the policy model can recommend top-k developers for a posted challenge of a customer.

We describe each component in detail in the rest of the section.

3.2 Data Extractor

3.2.1 Data Preparation

To obtain the developer and challenge data from Topcoder, we have developed a spider program to crawl the online traces of developer activities from January 2009 to February 2018. We eventually obtained Topcoder data containing 46,000 developers and 29 types of challenges, which is the largest Topcoder data used in studies on this topic as far as we know. We treat each type of challenge as a dataset. We remove the datasets that contain fewer than 10 winners as they are unpopular. The remaining 11 datasets are worth studying, which are shown in Table 1.

In Table 1, the Reg, Sub, and Win represent the number of developers with registration, submission, or winning history respectively. We filtered away around 36,000 incomplete challenges (i.e. the challenges that failed without winners or were canceled by the challenge organizers). In total, we have got 18,856 completed challenges, involving 41,827 registrants, 9,558 submitters, and 5,014 winners.

There are always some types of challenges that are more popular than others. For example, in Table 1, the three biggest datasets are Code, Assembly, and First2Finish, which contain more challenges and more developers than the other datasets. To reduce the data sparsity and improve recommendation accuracy, we cluster each large dataset into small ones. More specifically, we apply the k-means algorithm to cluster the challenges based on their contents. Then we apply our model to each cluster. According to our experiments, to obtain satisfactory clustering effect, k was finally set to 4, 4 and 8 for the three datasets (*Code*, *Assembly* and *First2Finish*) respectively.

In challenge-based CSD, only a small percentage of developers can eventually win the challenge. According to our statistics, in over 90% of all challenges, the number of

Table 1 The Topcoder datasets used in this study

Dataset (challenge type)	Challenges	Reg	Sub	Win
Conceptualization	243	1031	158	67
Content creation	106	995	163	66
Assembly	3437	3331	689	322
Test suites	142	523	100	60
UI prototype	1240	2591	450	124
Bug hunt	1285	1538	272	142
Code	3601	18786	5493	2999
First2Finish	6522	9802	1551	976
Design	753	598	140	61
Architecture	788	983	137	60
Development	739	1649	405	137

developers that actually registered with a specific challenge is no more than 100, while there are nearly 4000 developers having a winning history in total. Therefore, the winning data is highly imbalanced. To ease training, we balance the training set by oversampling. More specially, we apply ADASYN (He et al. 2008), which is an improved version of the SMOTE method (Chawla et al. 2002), to balance the data classes.

3.2.2 Developer Influence Graph (DIG)

Archak (2010) observed the phenomenon that the registration of competitive developers might deter the participation of others, while some developers are always willing to participate in a challenge with other developers. Although the empirical study conducted by Archak (2010) showed that there exits interaction influence between developers in the challenges, they did not propose a method to measure this influence. In order to quantify the developers' interaction in a CSD platform, in this work, we propose to build a directed graph DIG (Developer Influence Graph), which models the influence between two developers in a given challenge based on their previous histories. The graph is illustrated in Fig. 3, where the edge annotated with Influence Ratio (IR, defined in Eq. 1) indicates the fraction of common participant history with respect to a developer. According to Eq. 1, a larger $IR_{A,B}$ means that developer A has less "deter" or competence influence on developer B. The term $history_{A,B}$ represents the number of challenges participated by both developers A and B. The term $history_B$ indicates the number of challenges that developer B has participated in. Essentially, IR is defined on the basis of the confidence level of the association rule mining, which measures the influence of two developers statistically. In practice, the demographics attributes of developers may also affect their behaviour, which will be investigated in future work. We build three DIGs on registration, submission, or winning history, respectively. For example, when building DIG on registration, we count the $|challenges|$ that developer A and B both registered as $history_{A,B}$ and $|challenges|$ that developer B

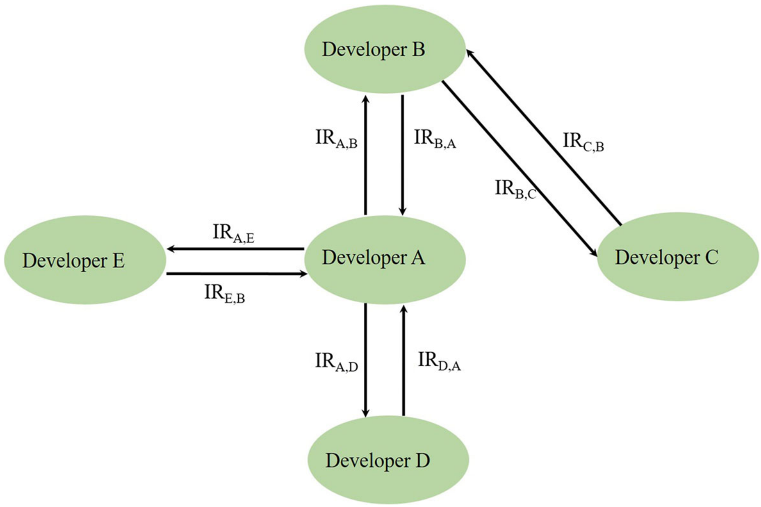


Fig. 3 Illustration of a DIG

registered as $history_B$. And DIG on submission, we count for submitted challenges and it is similar for DIG on winning.

$$IR_{A,B} = \frac{history_{A,B}}{history_B}. \tag{1}$$

Having constructed DIGs, we then apply the PageRank algorithm (Wang et al. 2016) to generate a normalized rank score for each node, which indicates the influence of the corresponding developer.

3.2.3 Feature Extraction

In our work, we identify and encode the following features:

- Challenge Features: We collect challenge-related features and encode them. In order to efficiently represent the challenges, we consider the textual descriptions of a challenge, the required programming languages techniques, challenge posting date, the number of days the challenge lasts, the total rewards of the challenge (prizes), and the difficulty of a challenge. To obtain a deeper understanding of a challenge, we encode the requirements of a challenge using Paragraph Vector (Le and Mikolov 2014), a state-of-the-art technique for natural language processing. We first select the titles and requirement descriptions of historical challenges and train a Paragraph Vector model. Then for the current challenge, we apply the Paragraph Vector model to transform the textual contents into a vector representation. The Paragraph Vector model considers both semantics and the order of words and can better represent the contents of a challenge. Following the related work (Fu et al. 2017), we encode the techniques and programming languages required by a challenge using one-hot feature encoding (Goodfellow et al. 2016) because they are discrete terms. Besides, we use the difficulty parameter D proposed in Wang et al. (2017), which is a synthetic normalized parameter that indicates the difficulty of a challenge. Specifically, D is calculated by combining four factors including the duration of a challenge, the amount of the prize, the number of registrants and the reliability bonus, and all the four factors are positively correlated with the difficulty parameter D. A summary of the challenge features is shown in Table 2. The number of dimensions that is used to encode each feature is also given(e.g. title(20) means the title feature dimension is 20). After concatenating all the features, we get a 130-dimension challenge feature vector.
- Developer Features: In order to efficiently represent a developer, we consider three types of features, which include developer intrinsic features (such as skills, member

Table 2 Challenge feature encoding

Features	Description
Languages (18)	one-hot encoding of the programming language
Techniques (48)	one-hot encoding of the technique used
Title (20)	a title vector encoded by Paragraph Vector
Requirements (40)	a requirement vector encoded by Paragraph Vector
Posting date (1)	the time when the challenge is posted
Duration (1)	the number of days the challenge lasts
Prizes (1)	the award offered by the customer
Difficulty (1)	difficulty of the challenge

age, historical features in registration, submission, winning and performance), challenge match features (such as language MD and technique MD), and interaction influential features extracted from DIG (such as registration rank, submission rank and winning rank). We extract four kinds of history data for developers, which are registration, submission, winning, and performance history. The registration history contains registration frequency (the number of challenges the developer has registered with) and the recency (the number of days since last registration). The submission history and winning history contain similar frequency and recency features. For those without corresponding history, we set the recency to infinite and the frequency to 0. The performance features consist of last rank and last score, which refer to the ranking and the score of the developers in the last challenge they participated in. We also encode the skills of developers using the one-hot encoding method. Besides, we compute the match degree between a developer's skills and the techniques and languages required by a challenge using the MD metric defined in Eq. 2. For example, if a challenge requires C# and JAVA and the skills of a developer contain JAVA and JS, then the $MD = 1/2 = 0.5$. In essence, the MD metric characterizes the matching degree between skills and requirements, which is also used in psychology (Edwards and Van Harrison 1993) and software engineering community (Hauff and Gousios 2015). In our work, the Topcoder platform provides tags to describe developer skills and challenge requirements, which helps us define the MD metric. For each challenge, we also build a DIG for all developers based on their registration, submission, and winning history, respectively. For each DIG, we obtain the PageRank score for each developer. A summary of developer features is shown in Table 3. The number of dimensions that is used to encode each feature is also given. We concatenate all the features to get a 60-dimension developer feature vector.

$$MD = \frac{sharedSkills_{(developer,challenge)}}{allRequirements_{(challenge)}}. \tag{2}$$

– Input data construction: For each posted challenge, we concatenate the 60-dimension developer feature vector with the 130-dimension challenge feature vector. The concatenation forms an input instance for the base predictors. For model training, we label each instance with the status (registered, submitted, or won). E.g. if a developer

Table 3 Developer feature encoding

Features	Description
Skills (46)	one-hot encoding of the skills
Member age (1)	days the developer becomes a member of Topcoder
Technique MD (1)	techniques match degree
Language MD (1)	languages match degree
Registration (2)	registration frequency and recency
Submission (2)	submission frequency and recency
Winning (2)	winning frequency and recency
Performance (2)	the score and rank in the last challenge
Registration rank (1)	the PageRank score in DIG on registration history
Submission rank (1)	the PageRank score in DIG on submission history
Winning rank (1)	the PageRank score in DIG on winning history

registers with the posted challenge, we assign 1 as the label; else we assign 0. The construction flow is illustrated in Fig. 4. Suppose there are m developers and n challenges, we encode m developer features and concatenate each vector with the challenge feature vector. Finally, we obtain $m \times n$ input vectors for training the model.

3.3 Base Predictors

Our policy model contains three base predictors (registration predictor, submission predictor, and winning predictor). Each predictor contains the following three base machine learning algorithms:

- ExtraTrees (Geurts et al. 2006), which is a bagging machine learning algorithm that is similar to the Random Forest algorithm (Breiman 2001). However, it selects the splitting attribute more randomly and performs better than Random Forest when there are many attributes. In our work, We use the ExtraTrees implementation of the scikit-learn package (Pedregosa et al. 2011).
- XGBoost (Chen and Guestrin 2016), which is a boosting algorithm that utilizes the second-order derivative of the error to conduct its boosting stages to avoid local optima. In our work, we use a scalable implementation of XGBoost at Chen et al. ().
- Neural network (Hinton and Salakhutdinov 2006), which is good at discovering hidden relations in data. The network structure we used is a 3-layer dense network that contains a 64-unit hidden layer 1, a 32-unit hidden layer 2, and an output layer. The activation function of hidden layers is ReLu (Goodfellow et al. 2016). We use a Softmax function in the output layer so that the output is in the range of 0 and 1. In our work, we use the Keras package (Chollet et al. 2015) to implement the neural network in Tensorflow (Abadi et al. 2016).

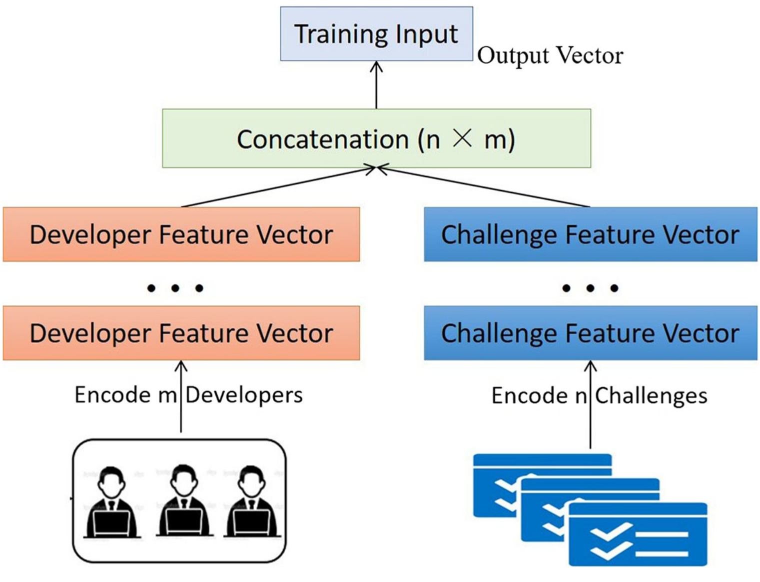


Fig. 4 Input data construction

The errors of a prediction model always contain three parts: bias, variance, and random error, which vary across different datasets (Valentini and Dietterich 2002; Domingos 2000). This intrinsic characteristic of the machine learning model results in its instability among different data. Except for the random error, both bias and variance can be eliminated or decreased via proper modeling approach. According to the theories of the above three algorithms, each of them has different inductive bias space which can fit well with some data instances but may not fit well with other instances (Sanjana and Tenenbaum 2003; Navarro et al. 2012). Therefore, we utilize these three algorithms to reduce the errors caused by biased inductive assumption space. Note that our framework is generic and other machine learning algorithms can be always incorporated.

We also utilize the Grid Search tool provided by the scikit-learn package (Pedregosa et al. 2011) to tune the hyper-parameters of machine learning algorithms. Grid Search is a simple case of hyper-parameter optimization (Hazan et al. 2017). Besides, we use Tensorflow (Abadi et al. 2016) as the backend to leverage the GPU resource for improving the runtime performance of the base predictors.

3.4 Meta-Learning Based Policy Model

3.4.1 Meta-Learning

Our proposed approach is based on meta-learning. Meta-learning aims at "learning to learn" (Metalearning 2009), which can automatically improve the performance of existing learning algorithms or induce the learning algorithms. Recently, it has been successfully used for algorithm recommendation (Cunha et al. 2018), hyper-parameter tuning (Hazan et al. 2017), and neural network optimization (Munkhdalai and Yu 2017), etc.

A typical meta-learning approach to algorithm recommendation (Cui et al. 2016; Al-Shedivat et al. 2017) consists of four spaces, namely problem space, meta-feature space, performance space, and algorithm space. The problem space includes the datasets of learning instances. The feature space is an abstract representation of the instances in the problem space. The algorithm space contains candidate algorithms in a given context, and the performance space is performance measurement of algorithms. The main goal is to select the best performing algorithm in the spaces. Formally, for a given problem instance $x \in P$, with features $f(x) \in F$, find the selection mapping $S(f(x))$ into the algorithm space A, such that the selected algorithm $\alpha \in A$ maximizes the performance mapping $y(\alpha(x)) \in Y$ (Rice 1976).

In our work, we use the input data instances as the problem space. The three machine learning algorithms described in previous section form the algorithm space. The meta-feature space is constructed by the possible choices of base algorithms and threshold parameters for all the base predictors (registration, submission, and winner). We evaluate the model using accuracy metrics and form the performance space. We select the best performing algorithm and thresholds given the spaces.

3.4.2 Tuning the Policy Model through Meta-Learning

The policy model is the central part of the system. To predict winners, our model needs the knowledge about developers' registration and submission behavior, which is provided by the registration and the submission predictors, respectively. A general structure of the policy model is illustrated in Fig. 2, where $P(Win)$ refers to the possibility that a developer will win a challenge. Our goal is to learn a sequence of predictions that can achieve the best winning prediction accuracy.

Algorithm 1 Meta-learning policy model.

Input: $perf\_metric$: performance metric; fsp: meta-feature space; s: step size; $policyModel$: the policy model framework; $data$: training data;
Output: $mf*$: meta feature instance under best performance;

$top\_R, top\_S, alg1, alg2, ..., aglN = fsp.getMetaFeatures()$
$algset_{meta-feature} = set(alg1, alg2, ..., aglN)$
$thresholdset_{meta-feature} = set(s, 2 * s, 3 * s, ..., 1)$
initialize $regModels$ with $algset_{meta-feature}$
initialize $subModels$ with $algset_{meta-feature}$
initialize $winModels$ with $algset_{meta-feature}$
% train meta models
traindata=data.getTrain()
for each $model \in regModels$ **do**
    $model.train(traindata)$;
end for
for each $model \in subModels$ **do**
    $model.train(traindata)$;
end for
for each $model \in winModels$ **do**
    $model.train(traindata)$;
end for
initialize top_R, top_S with $thresholdset_{meta-feature}$
$meta\_feature\_size = algset_{meta-feature}.size^N * thresholdset_{meta-feature}.size^2$
$grid = newvector(size = meta\_feature\_size)$
$combinationSet = enumerate(top\_R, top\_S, regModels, subModels, winModels)$
for each $(t1, t2, reg, sub, win) \in combinationSet$ **do**
    $mf =< t1, t2, reg, sub, win >$
    $grid.insert(mf)$;
end for
% search for the optimal meta feature under $perf\_metric$
$validatedata = data.getValidate())$
$bestPerf = 0$
$mf* = Null$
for each $i \in [1, meta\_feature\_size]$ **do**
    $mf = grid.get(i)$
    initialize an $policyModel$ instance with mf
    $Y = policyModel.predict(validatedata)$
    $perf = perf\_metric(validatedata, Y)$
    if $perf > bestPerf$ **then**
        $bestPerf = perf$
        $mf* = mf$
    end if
end for
return $mf*$

In order to achieve the best winning prediction accuracy (measured in terms of performance metric), we select an optimal combination of threshold parameters and algorithms through meta-learning (Cunha et al. 2018; Metalearning 2009). As we have three predictors and each of them contains three base machine learning algorithms, we have 3*3*3 possible combinations of the algorithms. The top_R and top_S are the threshold parameters that are used by registration and submission predictors respectively and influence the final result. For each of top_R and top_S, we consider their values ranging from 0 to 1, with a step of 0.01. Therefore, we have 100*100 possible choices of the threshold parameters. Then we build a 5-dimension cube with each dimension representing one instance of meta-feature. In total, the size of the meta-feature space is (3*3*3*100*100). We apply a search-based method to find the optimal combination of basic algorithms and threshold parameters that can achieve the best prediction performance. As the search space is not very large, we apply *Grid Search* to exhaustively search the space. We use the top 3, top 5, top 10 accuracy and MRR (Avazpour et al. 2014; Powers 2007; Aggarwal et al. 2016) as performance metrics to guide the search process.

Having finished the training, we select the optimal setting of meta-features that achieves the best winning prediction performance as the final setting for the policy model. The whole process is illustrated in Algorithm 1. In essence, the meta-learning method regards the learning context as the meta-features and evaluates them with respect to the performance measure. The learning context that maximizes the recommendation performance of the entire policy model is selected.

3.4.3 Using the Tuned Policy Model

Having tuned an optimal policy model, for a new challenge, we can apply the model to obtain a list of recommended developers. Given a set of developers, the model filters out the developers who are unlikely to register with and submit to the challenge, and recommends a list of developers ordered by their probability of winning the challenge. For a large dataset (e.g. Assembly, First2Finish, Code) that is divided into clusters (Section 3.2), when a new challenge comes, we first assign it to a cluster and then use the policy model built for that cluster to recommend developers.

4 Evaluation

In this section, we evaluate the proposed approach. We focus on the following research questions:

RQ1: Can the Proposed Developer Recommendation Approach Outperform the Baseline Methods? This RQ evaluates the overall effectiveness of the proposed approach and compares it with the performance of three baseline methods described in Section 2.2. The three baseline methods are CBC (Fu et al. 2017), CrowdRex (Mao et al. 2015), and DCW-DS (Yang et al. 2016). These methods also extract features from challenges and developers and use a machine learning algorithm for prediction. The CBC and CrowdRex methods treat winner prediction as a multi-label classification problem. The DCW-DS method helps developers estimate their roles (winner, submitter, or quitter) in a given challenge, and formulates the problem as a single-label 3-value classification problem.

To enable the comparison with DCW-DS, we make prediction for all the developers to see whether or not they could be a winner of a given challenge. In our work, we implement the three baseline methods, which can be accessed online.[6] Among the three baselines, CBC (Fu et al. 2017) is one of the works of our research teams; while for CrowdRex (Mao et al. 2015)and DCW-DS (Yang et al. 2016), our implementation achieves similar results as described in the original papers following their processing instructions. To achieve fair comparison, we apply Grid Search to all the baseline methods (using the scikit-learn wrapper) so that we always compare with the baseline method with the best-performing parameters. Besides, we do not filter developers with fewer than 5 winning records in the experiment.

RQ2: Is the Proposed Meta-Learning Based Policy Model Effective? This RQ evaluates the effectiveness of the meta-learning based policy model, which is the core part of the proposed approach. To evaluate the policy model, we compare it with the *winning predictor model*, which directly predicts winner without using the policy model. That is, we skip the registration and submission predictions (by setting the two parameters top_R and top_S to 100%) and use the output of $WinningPredictor$ directly to recommend developers for a given challenge. The rest are the same as the policy model.

As stated in Section 3.3, our meta-learning based policy model utilizes three basic algorithms, namely Neural Network, ExtraTrees, and XGBoost. In this RQ, we also compare the performance of the policy model with the performance of the three individual base algorithms.

RQ3: How do Different Features Affect the Performance of Our Model? We have proposed a set of features in Tables 2 and 3 for recommending reliable developers and we need to understand the contribution of those features to the effectiveness of our Policy-Model. Therefore we conducted an ablation study of the effect of different features on the performance of the PolicyModel. We studied the following feature groups: 1) the technique-related features of a challenge including languages and techniques, 2) the contents of a challenge including title and requirement, 3) the time related features including posting date and duration, 4) the features directly affecting the participation of developers including prizes and difficulty, 5) the features about developers' skills including skills, member age, technique MD and language MD, 6) the features about the participation history of a developer including registration, submission, winning and performance, 7) the ranking features of a developer obtained from DIG including registration rank, submission rank and winning rank. We tested the model performance by removing the one of the listed feature groups.

4.1 Experimental Setting

For each Topcoder dataset (i.e., each type of challenge such as Conceptualization and Bug Hunt), we firstly order all its challenges by the posting date from the oldest to the newest. The dataset is then split into three parts: the first 70% of the oldest challenges are used for training, the following 10% of challenges are used for validation, and the newest 20% of challenges are used for testing. Our policy model building consists of two process which are base predictor training using training dataset and meta-learning based policy model tuning using validating set. As described in Section 3.2, to facilitate the training of classification models, we balance the percentage of winner and non-winner through oversampling. Unlike the construction of the training set, we use the original distribution of data and do not perform oversampling in testing. For each challenge in the test set, our policy model

[6]https://github.com/zhangzhenyu13/CSDMetalearningRS/tree/master/Baselines

recommends the top k developers from a set of candidate developers, who are comprised of: 1) the developers who have winning history in this challenge type, including those whose winning records occurs only in test set; 2) a number of randomly selected Topcoder developers who have no winning history (in our experiments, the number of randomly selected developers is the same as the number of developers who have winning history). We will evaluate if the proposed approach can recommend the correct developers (winners) for the challenge, and if developers who never won the challenge before could still be recommended.

4.2 Evaluation Metrics

To evaluate the performance of the meta-learning based policy model, we leverage the $accuracy$ metric that is also used in related work (Mao et al. 2015; Fu et al. 2017). If the top k results in the recommended list contain the actual winners (usually 1-2) of the challenge, we consider the recommendation effective and calculate the percentage of effective recommendations for all challenges. Assume that we have N challenges and each is recommended with a list of developers, our accuracy metric is defined in Eq. 3, where N is the number of challenges in the test set and hit denotes whether the top k list contains the actual winners ($winners_n$) of the challenge (1 if an actual winner of the n^{th} challenge is in the top k list). Besides, we also apply another commonly-used metric Mean Reciprocal Rank (MRR) (Chowdhury and Soboroff 2002), which is the average of the reciprocal ranks of results of N challenges. The reciprocal rank of one recommendation list is the inverse of the rank of the first hit result (denoted as $Frank_n$). The higher the MRR, the better the model performance.

$$Acc@k = \frac{1}{N}\sum_{n=1}^{N} hit(winners_n, k) \tag{3}$$

$$MRR = \frac{1}{N}\sum_{n=1}^{N} \frac{1}{Frank_n} \tag{4}$$

Note: as described in Section 3.2.1, for a challenge of *Code*, *Assembly* or *First2Finish*, the recommendation will be made on the corresponding cluster. Then for each of the three datasets of *Code*, *Assembly* and *First2Finish*, we collect the ACC@k and MRR results of all the clusters, and report the weighted average. The weight is the percentage of challenges in each cluster. For example, there are $n_1, n_2, n_3\, and\, n_4$ challenges in the 4 clusters of *Assembly*, thus the weight of each cluster of *Assembly* is $n_i/\sum_1^4 n_i$ $(1 \le i \le 4)$.

4.3 Experimental Results

4.3.1 RQ1: Can the Proposed Developer Recommendation Approach Outperform the Baseline Methods?

Table 4 shows the recommendation accuracy of the proposed approach. The accuracy for top-3 recommendation ranges from 22.4% to 84.8%, with an average of 46.7%. The accuracy for top-5 recommendation ranges from 30.1% to 91.1%, with an average of 57.1%. The accuracy for top-10 recommendation ranges from 30.6% to 91.1%, with an average of 58.1%. Compared to the performance of baselines which are shown in Tables 5, 6 and 7, these results are significantly better than those achieved by the three baseline methods.

Table 4 The performance of PolicyModel

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.596	0.702	0.705	0.397
Test suites	0.64	0.76	0.76	0.4
Content creation	0.238	0.333	0.365	0.21
Conceptualization	0.553	0.702	0.703	0.227
Design	0.406	0.486	0.502	0.213
Development	0.4	0.493	0.511	0.257
UI prototype	0.463	0.598	0.615	0.371
Bug Hunt	0.848	0.911	0.911	0.759
Code	0.224	0.301	0.306	0.11
First2Finish	0.322	0.445	0.457	0.137
Assembly	0.442	0.545	0.552	0.3
Average	0.467	0.571	0.581	0.312

Table 5 The performance of CBC

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0	0	0	0.019
Test suites	0.04	0.04	0.12	0.056
Content creation	0	0	0	0.021
Conceptualization	0	0	0	0.026
Design	0.232	0.341	0.478	0.177
Development	0	0	0	0.013
UI prototype	0.336	0.377	0.622	0.357
Bug hunt	0.687	0.758	0.83	0.828
Code	0.025	0.029	0.037	0.035
First2Finish	0.018	0.023	0.068	0.03
Assembly	0.237	0.304	0.35	0.211
Average	0.143	0.17	0.228	0.161

Table 6 The performance of CrowdRex

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.006	0.007	0.338	0.071
Test suites	0	0	0	0.08
Content creation	0	0	0.095	0.044
Conceptualization	0.234	0.255	0.319	0.265
Design	0.079	0.152	0.347	0.124
Development	0.035	0.042	0.085	0.044
UI prototype	0.07	0.07	0.111	0.093
Bug hunt	0.312	0.321	0.366	0.411
Code	0.002	0.009	0.026	0.022
First2Finish	0.035	0.047	0.068	0.043
Assembly	0.115	0.134	0.168	0.107
Average	0.081	0.094	0.175	0.119

Table 7 The performance of DCW-DS

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.026	0.066	0.258	0.076
Test suites	0	0.04	0.36	0.067
Content creation	0	0	0	0.068
Conceptualization	0.17	0.277	0.362	0.219
Design	0.079	0.123	0.289	0.099
Development	0.186	0.236	0.271	0.129
UI prototype	0	0.008	0.016	0.025
Bug Hunt	0	0	0	0.033
Code	0.036	0.048	0.079	0.067
First2Finish	0.004	0.013	0.045	0.022
Assembly	0.161	0.215	0.28	0.16
Average	0.06	0.093	0.178	0.088

The average improvement of PolicyModel over the baseline methods is about three to five times. Table 8 shows the average performance of the baselines and the policy model side by side. Figure 5 also shows the MRR boxplots for all the comparative methods. The baseline methods can perform well for some challenges (refer to the outliers of the box-plot), while the average results are all lower than the $PolicyModel$. Clearly, the proposed approach achieves better overall accuracy in terms of MRR. We also use the experiment to help analyze why the baseline methods perform less satisfactory. The baseline methods recommend developers who have several historical winning records and filter away those developers with a few (1 or 2) winning records and the corresponding challenges. Thus the baselines can only recommend skillful developers and perform well on a subset of the datasets. However, in reality, many challenges are won by less skillful winners, thus when applying the baseline methods to the actual, complete datasets, their performance is less satisfactory. The major reason is due to baselines' limitation of only including developers with at least five winning records in the training data. However, we even get a worse result if we ignore this limitation. Because it significantly expands the developer set, the data becomes at least 10 times more sparse, which makes recommendation very challenging. The experiment results show that it is quite hard to overcome the difficulty of this limitation. Our policy model performs better than baseline methods because we leverage the concept of meta-learning to extract registration, submission, and winning meta-features via the meta-models. The meta models contain policy knowledge so that we can build a more accurate and general model.

4.3.2 RQ2: Is the Proposed Meta-Learning Based Policy Model Effective?

In order to evaluate the effectiveness of policy model, we test the performance of winner predictor alone to demonstrate the necessity of registration and submission predictors. We

Table 8 The performance of baseline methods and policy model

Methods	Acc@3	Acc@5	Acc@10	MRR
CBC	0.143	0.17	0.228	0.161
CrodRex	0.081	0.094	0.175	0.119
DCW-DS	0.06	0.093	0.178	0.088
PolicyModel	0.467	0.571	0.581	0.312

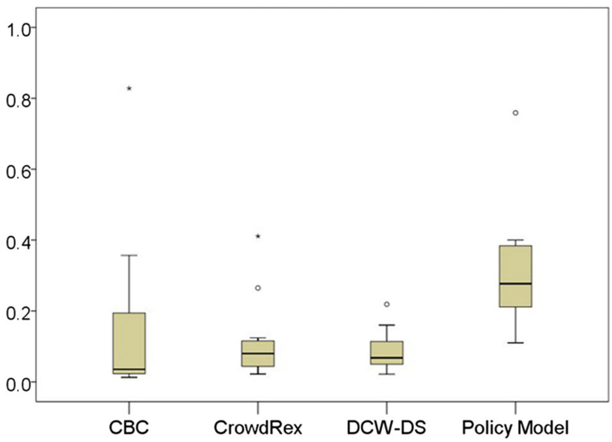


Fig. 5 The MRR of baseline methods and policy model

also test the performance of the three base machine learning algorithms to demonstrate the necessity of meta-learning. The results are shown in Tables 9, 10, 11 and 12. Compared with the result of winning predictor (which predicts winner directly without using the policy model), the policy model improves the average top 3, top 5, and top 10 accuracy by 0.21, 0.27, and 0.21, respectively. The experiment results confirm the effectiveness of our meta-learning based policy model. The winning predictor component performs poorly in some datasets because it contains no knowledge for registration and submission status. Therefore, those who did not register with or submit to the challenge may be wrongly predicted as winners. The proposed policy model can improve the performance because it can predict developer's registration and submission behavior when there is no observed registration or submission status.

In our meta-learning based policy model, we use three base algorithms as meta-features (Neural Network, ExtraTrees, and XGBoost). We evaluate the effectiveness of recommendation using each individual algorithm alone. The results are also given in Tables 9, 10 and 11. Table 13 shows the average performance of the base algorithms and the policy

Table 9 The performance of NeuralNetwork

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.43	0.43	0.444	0.306
Test suites	0.2	0.36	0.6	0.225
Content creation	0.095	0.095	0.095	0.126
Conceptualization	0.128	0.128	0.17	0.146
Design	0.08	0.08	0.116	0.089
Development	0.007	0.021	0.086	0.031
UI prototype	0	0.004	0.033	0.024
Bug hunt	0	0	0	0.024
Code	0.0141	0.0248	0.0503	0.03
First2Finish	0.004	0.01	0.066	0.024
Assembly	0.016	0.041	0.103	0.04
Average	0.089	0.108	0.16	0.209

Table 10 The performance of ExtraTrees

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.311	0.358	0.377	0.156
Test suites	0.28	0.28	0.28	0.134
Content creation	0.095	0.095	0.095	0.149
Conceptualization	0.277	0.319	0.382	0.246
Design	0.174	0.238	0.29	0.132
Development	0.257	0.271	0.279	0.159
UI prototype	0.258	0.307	0.41	0.307
Bug hunt	0.509	0.509	0.509	0.74
Code	0.0211	0.0457	0.0527	0.042
First2Finish	0.036	0.047	0.073	0.045
Assembly	0.085	0.112	0.157	0.087
Average	0.239	0.235	0.264	0.06

Table 11 The performance of XGBoost

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.026	0.066	0.258	0.076
Test suites	0	0.04	0.36	0.068
Content creation	0	0	0	0.068
Conceptualization	0.17	0.277	0.362	0.219
Design	0.08	0.123	0.289	0.098
Development	0.186	0.236	0.271	0.129
UI prototype	0	0.008	0.016	0.025
Bug hunt	0	0	0	0.033
Code	0.037	0.0487	0.079	0.067
First2Finish	0.004	0.013	0.046	0.022
Assembly	0.161	0.215	0.28	0.161
Average	0.06	0.093	0.178	0.088

Table 12 The performance of WinnerPredictor

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.43	0.43	0.444	0.306
Test suites	0.28	0.36	0.583	0.225
Content creation	0.143	0.19	0.19	0.149
Conceptualization	0.34	0.404	0.554	0.246
Design	0.174	0.283	0.29	0.132
Development	0.257	0.271	0.279	0.159
UI prototype	0.212	0.32	0.439	0.307
Bug hunt	0.723	0.741	0.759	0.74
Code	0.045	0.074	0.117	0.067
First2Finish	0.036	0.049	0.082	0.035
Assembly	0.161	0.215	0.28	0.161
Average	0.257	0.301	0.367	0.231

Table 13 The performance of base algorithms, winner predictor and policy model

Methods	Acc@3	Acc@5	Acc@10	MRR
Neural networks	0.089	0.108	0.16	0.209
Extratrees	0.239	0.0.235	0.264	0.06
XGBoost	0.06	0.093	0.178	0.088
WinnerPredictor	0.257	0.301	0.367	0.231
PolicyModel	0.467	0.571	0.581	0.312

model side by side. The box-plots of MRR results are shown in Fig. 6. Clearly, each of the three base algorithms achieves lower MRR than the proposed $PolicyModel$, because of the inductive assumptions they make. However, our policy model uses meta-learning to select the best algorithm for different data, thus the overall performance is greatly improved.

4.3.3 RQ3: How do Different Features Affect the Performance of Our Model?

Tables 14, 15 and 16 show the experimental results of the ablation study for different feature groups on the *Assembly*, *Test Suites* and *Bug Hunt* datasets, respectively. Compared with the performance of PolicyModel in Table 4, the importance of each feature group can be observed. The most important features are feature(5) and feature(6), which means that the skill related attributes and the participation history are very important to identify reliable developers. Feature(1) and feature(2) are also important as they specify the detailed information of the challenges which are usually considered by developers for selecting challenges. Developers will not be reliable if they are recommended to finish challenges that they are unwilling to choose. The ranking scores of developers (i.e. feature(7)) are also critical to the PolicyModel as they can measure the influential factors of developers in a challenge. Therefore DIG is useful. The incentive and difficult factors of a challenge (feature(4)) are not that important, the reason of which might be that many developers can hardly estimate the difficulty of a challenge and the motivation of many developers for participating in a challenge is to accumulate reputation or improve skills. And feature(3) have the least influence on the performance, which means that developers care less about posting date and challenge duration.

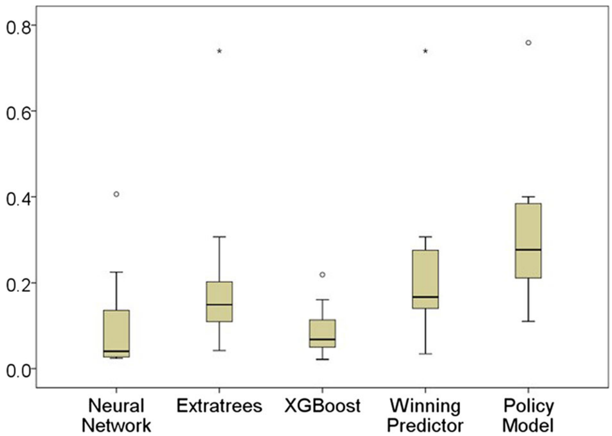


Fig. 6 The MRR of three base algorithms, winner predictor and policy model

Table 14 The ablation study of different features on *assembly*

Dataset	Acc@3	Acc@5	Acc@10	MRR
-feature(1)	0.411	0.516	0.535	0.287
-feature(2)	0.411	0.507	0.512	0.283
-feature(3)	0.442	0.543	0.549	0.3
-feature(4)	0.439	0.533	0.548	0.296
-feature(5)	0.372	0.397	0.408	0.247
-feature(6)	0.385	0.388	0.409	0.252
-feature(7)	0.413	0.508	0.52	0.288
All	0.442	0.545	0.552	0.3

Table 15 The ablation study of different features on *test suites*

Dataset	Acc@3	Acc@5	Acc@10	MRR
-feature(1)	0.536	0.617	0.634	0.349
-feature(2)	0.52	0.597	0.653	0.323
-feature(3)	0.571	0.635	0.678	0.377
-feature(4)	0.554	0.628	0.645	0.356
-feature(5)	0.429	0.465	0.496	0.318
-feature(6)	0.437	0.458	0.499	0.312
-feature(7)	0.545	0.598	0.644	0.385
All	0.64	0.76	0.76	0.4

Table 16 The ablation study of different features on *bug hunt*

Dataset	Acc@3	Acc@5	Acc@10	MRR
-feature(1)	0.768	0.819	0.821	0.633
-feature(2)	0.776	0.824	0.825	0.645
-feature(3)	0.826	0.893	0.897	0.734
-feature(4)	0.817	0.865	0.874	0.728
-feature(5)	0.739	0.796	0.815	0.597
-feature(6)	0.748	0.793	0.831	0.602
-feature(7)	0.783	0.816	0.836	0.667
All	0.848	0.911	0.911	0.759

In summary, we propose a study for exploring the influence of different features, where we have implemented a general method through applying masks to the input to eliminate the function of the masked features. The results show that the importance of different features is slightly different across datasets. Additionally, Calefato et al. (2018) have studied how to ask questions more effectively in Stack Overflow, and our study on the importance of the those features also show some insights for both developers and customers. Developers need to enhance their skills and participate more frequently and actively for winning a challenge in future. Customers can know which are the most important attributes when they post challenges.

5 Discussions

The experimental results described in the previous section show that our developer recommendation approach outperforms the existing ones. The results also show that the proposed meta-learning PolicyModel is effective. To better analyze the capacity of our recommendation approach, we consider the following problems:

- Can our model recommend new winners in CSDs?
- How does our model perform for the recommendation in each stage?

5.1 Support for Recommending New Developers

Our model is different from the existing methods for developer recommendation in CSD. As we have discussed, existing methods either consider only the developers who have sufficient winning records or assume the registration/submission status, which rule out the other developers to be recommended. The essential reason for existing methods to make such assumption is that they cannot handle the data sparsity well. We do not make any assumption about developer status (registration or submission). We build a PolicyModel to predict the developer status. Instead of using a fixed set of developers as labels to build a multi-label classification model, our PolicyModel outputs a probability value for each developer and ranks the developers by the probability values. In light of the limitation of existing methods, on the one hand our PolicyModel divides the recommendation process into three stages; on the other hand our model employ meta-learning to tune the learning parameters. The adoption of the three-stage recommendation filters away the irrelevant data step by step, thus

Table 17 The percentage of winners who do not appear in the training phase

Dataset	*Missing*	Dataset	*Missing*
Architecture	55%	Development	70.8%
Conceptualization	82%	Bug Hunt	85.2%
Content creation	92.4%	Design	47.5%
Assembly	59.3%	Code	88.4%
Test suites	85%	First2Finish	78.6%
UI prototype	65.3%		

Table 18 The performance of our PolicyModel in registration stage

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.544	0.589	0.637	0.633
Test suites	0.448	0.522	0.726	0.274
Content creation	0.23	0.251	0.296	0.18
Conceptualization	0.577	0.629	0.781	0.451
Design	0.299	0.351	0.587	0.201
Development	0.407	0.56	0.729	0.318
UI prototype	0.579	0.833	0.833	0.618
Bug hunt	0.735	0.79	0.845	0.667
Code	0.212	0.294	0.388	0.153
First2Finish	0.466	0.563	0.751	0.415
Assembly	0.3	0.374	0.519	0.151
Average	0.436	0.523	0.644	0.369

reduce the data sparsity. At the same time, the meta-learning approach can fit the sparse data better by automatically tuning the parameters. Therefore our model can predict a developer as a potential winner according to the developer's participation history and current challenge requirements, despite that the developer has never won before. Table 17 shows that 47.5% to 92.4% of the winners are new developers who do not appear in the training phase (we use the term *Missing* to denote it). For example, there are 2999 winners in the Code dataset and only 347 (11.6%) winners appear in the training phase (in the training and validation sets). In fact, the test set contains relatively new members of Topcoder, who have fewer historical records than those in the training set. The results confirm that our model can recommend potential winners even though they have never won any challenge before.

5.2 Recommendation Performance in Each Stage

We further analyze the performance of our model for each stage. Tables 18, 19 and 20 show the experimental results for recommending registers, submitters and winners, respectively. For each challenge and all the candidate developers, we filtered away those that do not

Table 19 The performance of our PolicyModel in submission stage

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.915	0.954	0.993	0.87
Test suites	0.915	0.927	0.969	0.755
Content creation	0.762	0.857	0.915	0.514
Conceptualization	0.468	0.638	0.787	0.269
Design	0.768	0.949	0.96	0.595
Development	0.657	0.843	0.869	0.667
UI prototype	0.656	0.795	0.85	0.502
Bug hunt	0.67	0.777	0.893	0.285
Code	0.385	0.455	0.604	0.188
First2Finish	0.827	0.853	0.928	0.694
Assembly	0.809	0.828	0.889	0.496
Average	0.712	0.806	0.877	0.53

Table 20 The performance of our PolicyModel in winning stage

Dataset	Acc@3	Acc@5	Acc@10	MRR
Architecture	0.92	0.95	0.967	0.64
Test suites	0.945	0.96	0.999	0.557
Content creation	0.857	0.905	0.952	0.318
Conceptualization	0.723	0.83	0.936	0.305
Design	0.937	0.955	0.969	0.736
Development	0.921	0.95	0.957	0.699
UI prototype	0.905	0.948	0.988	0.82
Bug Hunt	0.901	0.96	0.994	0.815
Code	0.853	0.915	0.954	0.585
First2Finish	0.894	0.925	0.968	0.533
Assembly	0.892	0.937	0.991	0.6
Average	0.886	0.93	0.97	0.6

register with the challenge when evaluating the submission stage because we only consider whether a registered developer will make submission. Likely, we filtered away those developers that do not submit when evaluating the winning stage because we only consider whether a developer that submits can win. In the evaluation of registration stage, we did not filter away any developer as it is the first stage. The experimental results show that our model achieves 0.369, 0.53 and 0.6 MRR scores in average for registration, submission and winning stages, respectively. It is also worth mentioning that in the winning stage we obtain 0.886 Acc@3 score, 0.93 Acc@5 score and 0.97 Acc@10 score in average, which is mainly due to that many incompetent developers are filtered away in the two previous stages. Those with high winning frequency are very likely to win for a new challenge and therefore some previous works (Fu et al. 2017; Mao et al. 2015) filter away the developers without high winning frequency to improve the evaluation performance. However, such processing is biased because the statistics in Table 17 show that quite a few winners of a challenge are "first-time winners". The performance of the registration stage is not as good as that of the later stages because there are many developers to consider for the first stage. The DCW-DS in our baselines is based on the binary classification, which can be easily affected by the error accumulation of each stage as the pipeline is a direct combination of the predication results of each stage. In our PolicyModel, we adopt meta-learning to optimally combine the learners in the three stages, which significantly improves the recommendation performance.

6 Threats to Validity

We identify the following threats to validity:

- Subject selection bias. In our experiment, we only use 11 Topcoder datasets as experimental subjects. In fact, we collected 29 representative types of challenges posted between January 2009 and February 2018 in Topcoder. However, 18 of them contain fewer than 10 unique winners and a small number of challenges, therefore we discarded them. Among the remaining 11 types of challenges, 4 of them are also used in related work (Fu et al. 2017; Mao et al. 2015). Although our datasets are much larger than those used in related work, we will further reduce this threat by crawling more data from Topcoder in future. Furthermore, although Topcoder is a leading CSD platform, we will also seek to collect data from other CSD platforms to enhance the generalizability of our approach.

- Algorithm selection. In this paper, we choose 3 base machine learning algorithms (Neural Network, ExtraTrees, and XGBoost). Clearly, there are many other algorithms and it is unrealistic to test with all possible algorithm combinations. In this work, we purposely choose 3 algorithms that have different inductive biases that can complement each other. The selected 3 algorithms are widely used in industry and perform well in most cases. And our model is flexible to allow readers to incorporate other algorithms (, while we recommend to select at least one bagging and one boosting algorithms for their ability to reduce variance and bias respectively).
- Feature engineering. We have identified many features about challenges and developers to build our model, including the features used in related work. However, it is possible that there are other representative features. Furthermore, semantic features of textual descriptions (such as those identified through deep learning) could also be used. Systematic feature engineering will be studied in our future work.
- Benefits for CSD platforms. In this work, we conducted experiments to evaluate the effectiveness of our approach with the history datasets of Topcoder. Our model advance the state-of-the-art by removing the assumption of the number of winning records and the registration/submission status. However, merely recommending winners for posted challenges may discourage newcomers and less skillful developers, which can affect the long-term development of a CSD platform. We will investigate the benefits of our approach to real CSD platforms and obtain the feedback from real task requesters and developers in future work.
- Although Archak (2010) observed the interaction influence between developers, they did not propose a method to model such influence. Therefore, we propose the IR measure in this work (Section 3.2.2). Although our ablation study (Section 4.3.3) shows that DIG is useful, the IR measure has not been validated and the ground truth could differ from this conceptualization. In the future, we will consider obtaining the ground truth about the deterring relationship through a survey and compare the values obtained by measuring IR with the ground truth. We will also explore the construction of the DIG with different measures.

7 Related Work

7.1 Developer Recommendation in CSD

Recommendation system has been an active research topic in software engineering. Various methods have been proposed to recommend code reviewers (Hannebauer et al. 2016), bug-fixers (Anvik et al. 2006; Hu et al. 2014), question answerers (Choetkiertikul et al. 2015; Procaci et al. 2016), programming-related information (Ponzanelli et al. 2017), APIs (Yuan et al. 2018; Gu et al. 2016), etc. The proposed recommendation methods in existing work provide helpful background knowledge for studying the crowdsourcing developer recommendation. However, as each method is highly tuned for specific application scenarios and datasets, existing methods cannot be directly applied for Topcoder-like CSD platforms.

There is also much work on developer recommendation for CSD. For example, Fu et al. (2017) proposed a clustering based collaborative filtering classification model built using Naive Bayes algorithm, and formulated the winner prediction problem as a multi-label classification problem. Baba et al. (2016) proposed a crowdsourcing contest recommendation model for participation and winner prediction. However, their experiment results show that their model performs poorly when no participation information is available. We have also discussed and compared with the CBC (Fu et al. 2017), CrowdRex (Mao et al. 2015), and DCW-DS (Yang et al. 2016) work in this paper. Our experimental results show that the proposed meta-learning based policy model outperforms the related work.

There is also much research that studies crowd developers in CSD. For example, Alelyani and Yang (2016) conducted research to study the behavior of developers in Topcoder. They found many factors that can represent the developers' reliability. Saremi et al. (2017) performed an empirical study of crowd developers in Topcoder and investigated their availability and performance on the tasks. Abhinav et al. (2017) proposed a multidimensional assessment framework for hiring CSD workers. Dwarakanath et al. (2016) pointed out the untrustworthiness of some crowd developers, who could lead to task failure. Javadi Khasraghi and Aghaie (2014) investigated the relationship between developers' participation history and performance. In our work, we consider the features mentioned above and also add some new features (such as the Developer Influence Graph and the MD metric) based on the characteristics of CSD.

Instead of recommending developers, Karim et al. (2018) studied the problem of recommending tasks for crowdsourcing software developers by considering the exploitation and exploration motives of developers, and they proposed the EX^2 system that defines both the "LEARN" and "EARN" scores to characterize developers. Especially, with the "LEARN" score, EX^2 can make recommendation for the newcomers who even do not have any history in a CSD platform. Ye et al. (2018) also consider the skill learning requirements of crowdsourcing developers for recommending teammates in Kaggle. We believe the learning motive can be incorporated to complement our model for solving the cold-start problem.

7.2 Meta-Learning and Parameter Tuning

A meta-learning model is characterized by its capacity of learning from previous experiences and to adapt its bias dynamically conforming to the target domain (Brazdil et al. 2008). Meta-learning can also help build better models on small training datasets (Munkhdalai and Yu 2017). According to the work of Al-Shedivat et al. (2017), humans can leverage previously learned policies and apply them to new tasks. They leverage previously trained networks to store policies and apply them to build new models. Cui et al. (2016) proposed a meta-learning framework to recommend the most proper algorithms for the whole system accurately.

Recently, there are also some research on software defect prediction through meta-learning. For example, Tantithamthavorn et al. (2016) found that an automated parameter optimization technique named Caret can significantly enhance the performance of software defect prediction models. Porto et al. (2018) proposed and evaluated a meta-learning solution designed to automatically select and recommend the most suitable Cross-Project Defect Prediction (CPDP) method for a project. They found that the meta-learning approach can leverage previously experiences and recommend methods dynamically. In our work, we apply meta-learning to tune a policy model for developer recommendation.

8 Conclusion

In this paper, we propose a meta-learning based PolicyModel, which can recommend suitable crowd developers for crowd-sourced tasks (challenges). Our approach can recommend developers regardless of their registration or submission status, which is more realistic in practice. Through meta-learning, we build a PolicyModel to filter out the developers who are unlikely to register with a challenge and submit work, and find the reliable developers who are more likely to win the challenge. Our experiments on Topcoder datasets confirm the effectiveness of the proposed approach. Our tool and experimental data is publicly available at: https://github.com/zhangzhenyu13/CSDMetalearningRS.

In the future, we will experiment with other CSD platforms and the ecosystem of CSD (Li et al. 2015) to understand to what extent our approach can benefit real CSD. We also plan to build a challenge recommendation system considering the relationship among challenges. Such a system can also facilitate timely completion of the challenges posted by customers. We will investigate the benefits of our approach to real CSD platforms and obtain the feedback from real task requesters and developers in future work.

Acknowledgements This work was supported partly by National Key Research and Development Program of China under Grant No.2016YFB1000804, partly by National Natural Science Foundation under Grant No.(61702024, 61828201, 61421003).

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker PA, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) Tensorflow: a system for large-scale machine learning. In: 12th USENIX Symposium on operating systems design and implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016, pp 265–283

Abhinav K, Dubey A, Jain S, Virdi G, Kass A, Mehta M (2017) Crowdadvisor: a framework for freelancer assessment in online marketplace. In: Proceedings of the 39th international conference on software engineering: software engineering in practice track. IEEE Press, pp 93–102

Aggarwal CC et al (2016) Recommender systems. Springer

Al-Shedivat M, Bansal T, Burda Y, Sutskever I, Mordatch I, Abbeel P (2017) Continuous adaptation via meta-learning in nonstationary and competitive environments, arXiv:1710.03641

Alelyani T, Yang Y (2016) Software crowdsourcing reliability: an empirical study on developers behavior. In: Proceedings of the 2nd international workshop on software analytics. ACM, pp 36–42

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th international conference on software engineering. ACM, pp 361–370

Archak N (2010) Money, glory and cheap talk: analyzing strategic behavior of contestants in simultaneous crowdsourcing contests on topcoder.com. In: Proceedings of the 19th international conference on World wide web. ACM, pp 21–30

Avazpour I, Pitakrat T, Grunske L, Grundy J (2014) Dimensions and metrics for evaluating recommendation systems, pp 245–273

Baba Y, Kinoshita K, Kashima H (2016) Participation recommendation system for crowdsourcing contests. Expert Syst Appl 58:174–183

Begel A, Herbsleb JD, Storey M-A (2012) The future of collaborative software development. In: Proceedings of the ACM 2012 conference on computer supported cooperative work companion. ACM, pp 17–18

Brazdil P, Carrier CG, Soares C, Vilalta R (2008) Metalearning: applications to data mining. Springer Science & Business Media

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Calefato F, Lanubile F, Novielli N (2018) How to ask for technical help? Evidence-based guidelines for writing questions on stack overflow. Inf Softw Technol 94:186–207

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357

Chen T, Guestrin C (2016) Xgboost: a scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. ACM, pp 785–794

Chen T, Benesty M et al A high performance implementation of xgboost. [Online]. Available: https://github.com/dmlc/xgboost/

Choetkiertikul M, Avery D, Dam HK, Tran T, Ghose AK (2015) Who will answer my question on stack overflow? In: 24th Australasian software engineering conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015, pp 155–164
Chollet F et al (2015) Keras. [Online]. Available: https://keras.io
Chowdhury A, Soboroff I (2002) Automatic evaluation of world wide web search services. In: Proceedings of the 25th annual international ACM SIGIR conference on research and development in information retrieval. ACM, pp 421–422
Cui C, Hu M, Weir JD, Wu T (2016) A recommendation system for meta-modeling: a meta-learning based approach. Expert Syst Appl 46:33–44
Cui Q, Wang S, Wang J, Hu Y, Wang Q, Li M (2017) Multi-objective crowd worker selection in crowdsourced testing. In: 29th International conference on software engineering and knowledge engineering (SEKE), pp 218–223
Cunha T, Soares C, de Carvalho AC (2018) Metalearning and recommender systems: a literature review and empirical study on the algorithm selection problem for collaborative filtering. Inform Sci 423:128–144
Domingos P (2000) A unified bias-variance decomposition. In: Proceedings of 17th international conference on machine learning, pp 231–238
Dubey A, Abhinav K, Virdi G (2017) A framework to preserve confidentiality in crowdsourced software development. In: Proceedings of the 39th international conference on software engineering companion. IEEE Press, pp 115–117
Dwarakanath A, Shrikanth N, Abhinav K, Kass A (2016) Trustworthiness in enterprise crowdsourcing: a taxonomy & evidence from data. In: Proceedings of the 38th international conference on software engineering companion. ACM, pp 41–50
Edwards JR, Van Harrison R (1993) Job demands and worker health: three-dimensional reexamination of the relationship between person-environment fit and strain. J Appl Psychol 78(4):628
Fu Y, Sun H, Ye L (2017) Competition-aware task routing for contest based crowdsourced software development. In: 2017 6th International workshop on software mining (SoftwareMining). IEEE, pp 32–39
Geurts P, Ernst D, Wehenkel L (2006) Extremely randomized trees. Mach Learn 63(1):3–42
Goodfellow I, Bengio Y, Courville A, Bengio Y (2016) Deep learning, vol 1. MIT Press, Cambridge
Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 631–642
Hannebauer C, Patalas M, Stünkel S, Gruhn V (2016) Automatically recommending code reviewers based on their expertise: an empirical comparison. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering. ACM, pp 99–110
Hasteer N, Nazir N, Bansal A, Murthy B (2016) Crowdsourcing software development: many benefits many concerns. Procedia Comput Sci 78:48–54
Hauff C, Gousios G (2015) Matching github developer profiles to job advertisements. In: 12th IEEE/ACM Working conference on mining software repositories, MSR 2015, Florence, Italy, May 16–17, 2015, pp 362–366
Hazan E, Klivans AR, Yuan Y (2017) Hyperparameter optimization: a spectral approach. arXiv:1706.00764
He H, Bai Y, Garcia EA, Li S (2008) Adasyn: adaptive synthetic sampling approach for imbalanced learning. In: IEEE International joint conference on neural networks, 2008. IJCNN 2008. (IEEE World congress on computational intelligence). IEEE, pp 1322–1328
Hinton GE, Salakhutdinov RR (2006) Reducing the dimensionality of data with neural networks. Science 313(5786):504–507
Hu H, Zhang H, Xuan J, Sun W (2014) Effective bug triage based on historical bug-fix information. In: 2014 IEEE 25th International symposium on in software reliability engineering (ISSRE). IEEE, pp 122–132
Javadi Khasraghi H, Aghaie A (2014) Crowdsourcing contests: understanding the effect of competitors' participation history on their performance. Behav Inf Technol 33(12):1383–1395
Karim MR, Yang Y, Messinger D, Ruhe G (2018) Learn or earn? - intelligent task recommendation for competitive crowdsourced software development. In: 51st Hawaii international conference on system sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018, pp 1–10
Khanfor A, Yang Y, Vesonder G, Ruhe G, Messinger D (2017) Failure prediction in crowdsourced software development. In: 2017 24th Asia-Pacific software engineering conference (APSEC). IEEE, pp 495–504
Le Q, Mikolov T (2014) Distributed representations of sentences and documents. In: International conference on machine learning, pp 1188–1196
Li W, Huhns MN, Tsai WT, Wu W (2015) Crowdsourcing: cloud-based software development. Springer Publishing Company, Incorporated

Mao K, Yang Y, Wang Q, Jia Y, Harman M (2015) Developer recommendation for crowdsourced software development tasks. In: 2015 IEEE symposium on service-oriented system engineering (SOSE). IEEE, pp 347–356

Mao K, Capra L, Harman M, Jia Y (2017) A survey of the use of crowdsourcing in software engineering. J Syst Softw 126:57–84

Metalearning (2009) Concepts and systems. Springer, Berlin, pp 1–10

Munkhdalai T, Yu H (2017) Meta networks, arXiv:1703.00837

Navarro DJ, Dry MJ, Lee MD (2012) Sampling assumptions in inductive generalization. Cognit Sci 36(2):187–223

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830

Ponzanelli L, Scalabrino S, Bavota G, Mocci A, Oliveto R, Di Penta M, Lanza M (2017) Supporting software developers with a holistic recommender system. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE). IEEE, pp 94–105

Porto F, Minku L, Mendes E, Simao A (2018) A systematic study of cross-project defect prediction with meta-learning, arXiv:1802.06025

Powers D (2007) Evaluation: from precision, recall and fmeasure to roc, informedness, markedness and correlation. J Mach Learn Technol 2(01):37–63

Procaci TB, Nunes BP, Nurmikko-Fuller T, Siqueira SWM (2016) Finding topical experts in question & answer communities. In: 16th IEEE International conference on advanced learning technologies, ICALT. Austin, TX, USA, July 25-28, 2016, pp 407–411

Rice JR (1976) The algorithm selection problem. Adv Comput 15:65–118

Sanjana NE, Tenenbaum JB (2003) Bayesian models of inductive generalization. In: Advances in neural information processing systems, pp 59–66

Saremi R, Yang Y (2015) Dynamic simulation of software workers and task completion. In: Proceedings of the second international workshop on crowdsourcing in software engineering. IEEE Press, pp 17–23

Saremi R, Yang Y, Ruhe G, Messinger D (2017) Leveraging crowdsourcing for team elasticity: an empirical evaluation at topcoder. In: 2017 IEEE/ACM 39th International conference on software engineering: software engineering in practice track (ICSE-SEIP). IEEE, pp 103–112

Stol K-J, Fitzgerald B (2014) Researching crowdsourcing software development: perspectives and concerns. In: Proceedings of the 1st international workshop on crowdsourcing in software engineering. ACM, pp 7–10

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Automated parameter optimization of classification techniques for defect prediction models. In: 2016 IEEE/ACM 38th International conference on software engineering (ICSE). IEEE, pp 321–332

Valentini G, Dietterich TG (2002) Bias—variance analysis and ensembles of svm. In: International workshop on multiple classifier systems. Springer, pp 222–231

Wang W, He D, Fang M (2016) An algorithm for evaluating the influence of micro-blog users. In: Proceedings of the 2016 international conference on intelligent information processing. ACM, p 14

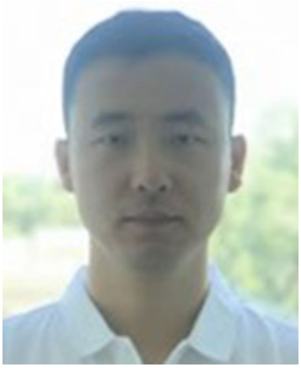
Wang Z, Sun H, Fu Y, Ye L (2017) Recommending crowdsourced software developers in consideration of skill improvement. In: 2017 32nd IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 717–722

Yang Y, Karim MR, Saremi R, Ruhe G (2016) Who should take this task?: dynamic decision support for crowd workers. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement. ACM, p 8

Ye L, Sun H, Wang X, Wang J (2018) Personalized teammate recommendation for crowdsourced software developers. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE. Montpellier, France, September 3-7, 2018, pp 808–813

Yuan W, Nguyen HH, Jiang L, Chen Y (2018) Libraryguru: api recommendation for android developers. In: Proceedings of the 40th international conference on software engineering: companion proceeedings. ACM, pp 364–365

Zanatta AL, Machado L, Steinmacher I (2018) Competence, collaboration, and time management: barriers and recommendations for crowdworkers. In: Proceedings of the 5th international workshop on crowd sourcing in software engineering. ACM, pp 9–16



Zhenyu Zhang is currently a master student of computer science in Beihang University. He received his BS degree from Tsinghua University in 2017 and will receive his MS degree from Beihang University in 2020. His research interest includes deep reinforcement learning, recommendation, question answering and software engineering.

Hailong Sun received the BS degree in computer science from Beijing Jiaotong University in 2001. He received the PhD degree in computer software and theory from Beihang University in 2008. He is an Associate Professor in the School of Computer Science and Engineering, Beihang University, Beijing, China. His research interests include intelligent software engineering, crowd intelligence/crowdsourcing, and distributed systems. He is a member of the IEEE and the ACM.

Hongyu Zhang is currently an associate professor with The University of Newcastle, Australia. Previously, he was a lead researcher at Microsoft Research Asia and an associate professor at Tsinghua University, China. He received his PhD degree from National University of Singapore in 2003. His research is in the area of Software Engineering, in particular, software analytics, testing, maintenance, metrics, and reuse. He has published more than 120 research papers in reputable international journals and conferences. He received two ACM Distinguished Paper awards. He has also served as a program committee member/track chair for many software engineering conferences. More information about him can be found at: https://sites.google.com/view/hongyu-zhang.