A Learning to Rank Framework for Developer Recommendation in Software Crowdsourcing

Jiangang Zhu  
School of Software  
Shanghai Jiao Tong University  
Shanghai, 200240, China  
jszigtws@sjtu.edu.cn

Beijun Shen  
School of Software  
Shanghai Jiao Tong University  
Shanghai, 200240, China  
bjshen@sjtu.edu.cn

Fanghuai Hu  
East China University of Science & Technology  
Shanghai, 200237, China  
xiaohuqi@163.com

Abstract—Recently, crowdsourcing has been widely used in many tasks that computers are not good at such as image recognition, entity resolution or some question answering tasks. A key feature of these tasks is that they are all simple tasks even decision making tasks. People can deal with these tasks with common sense knowledge. However, different from crowdsourcing in a general domain, software crowdsourcing is more complex. Only people with software developing skills can finish these tasks which could take a long time. Thus, an essential component of building a successful software crowdsourcing platform is effective developer recommendation, which matches a given task to the right crowdworkers. In order to solve this problem, in this paper, we propose a learning to rank framework. Specifically, we first propose a CRF-based model to extract criterias (i.e. skills and locations) from descriptions. Tasks characteristics learned from their descriptions and developer characteristic distributions extracted from their historical tasks are fed into learning to rank algorithms for developer recommendation together with some other features such as topic-based features. We have evaluated our approach on a large dataset crawled from a real-world software crowdsourcing platform. The experimental results show that our approach is feasible and effective.

Keywords—Software Crowdsourcing, Developer Recommendation, Software Engineering

I. INTRODUCTION

Thanks to its ability to combine scalable algorithm with the power of human intelligence, crowdsourcing techniques have been widely used as a solution of some computationally challenging tasks (e.g. taxonomy construction [15] and entity resolution [17], [18]). Many crowdsourcing platforms have arisen to perform paid crowdsourcing tasks such as Amazon Mechanical Turk\(^1\) (AMT), Topcoder\(^2\) and Zhubajie\(^3\).

Unfortunately, crowdsourcing on these platforms may yields relatively low-quality results due to its openness. This problem could be even more serious in software crowdsourcing. This is mainly because that software development is more professional compared with simple tasks from general domain (e.g. image recognition). The whole lifecycle of a software developing task is usually very long and the crowdsourcers need to pay relatively high rewards when the tasks are finished. In total, the cost (time and money) of software crowdsourcing is rather expensive. Thus, an effective developer recommendation component is very essential for building a successful software crowdsourcing platform.

There have been considerable amount of research works on developer recommendation in software engineering area. Most existing developer recommendation approaches are used in open-source communities to find the best developer to fix bugs or review codes. Regarding crowdsourcing research, Yudian Zheng et al. [27] proposed an optimization algorithm for Jury Selection Problem (JSP). Ju Fan et al. [7] introduced an adaptive crowdsourcing platform named iCroud. It on-the-fly estimates accuracies of a worker by evaluating her performance on the completed tasks, and predicts which tasks the worker is well acquainted with. Although existing methods perform well in simple crowdsourcing tasks and traditional developer recommendation problems, they may have limitations on more complicated crowdsourcing tasks such as software development tasks which require domain knowledge. The most related research is [27]. They proposed a detailed task and developer model and used a regression-based model to construct the score function for recommendation in software crowdsourcing. However, the proposed task and developer model is too idealistic because most software crowdsourcing platforms cannot provide such well-structured information. Thus, it is impractical for real-world applications. In this paper, we try to design a feasible and effective developer recommendation approach and evaluate our approach on a real-world dataset.

The problem is non-trivial and we address two main research challenges that arise in software crowdsourcing:

- Since many software crowdsourcing platforms use free text to describe tasks, it is difficult to find out the requirements of tasks (e.g. skill, location), which is essential for making right recommendations.
- There are different kinds of information which are indeed helpful for developer recommendation in crowdsourcing platform such as developers’ credit, historical tasks’ compatibility and location constraints. How to define a model to incorporate these factors together? How to make the model more flexible when adding more factors?

In order to solve above challenges, we propose a learning to rank based approach. Specifically, our approach leverages several features from different aspects to measure the compatibility between tasks and developers. Then a learning to rank

\(^1\)http://www.mturk.com  
\(^2\)http://www.topcoder.com/  
\(^3\)http://www.zhubajie.com/
algorithm is applied to rank the developers for each task. Our contributions mainly include:

- We propose a CRF-based approach to learn tasks characteristics from their descriptions and developer characteristic distributions from their historical tasks. A taxonomy-based skill matching algorithm is applied to computing the skill compatibility. Moreover, we also use the description of each task to learn a topic representation by applying Latent Dirichlet Allocation (LDA) [2]. Together with the normalized evaluation scores of each developer provided by the crowdsourcing platform, our proposed approach can combine evidences from the skill compatibility, the implicit topic relevance and the evaluation scores.

- We propose a unified model that incorporates these features for effective developer recommendation. Specifically, we treat the developer recommendation as a learning to rank problem to solve. Several learning to rank algorithms are applied to rank the developers for each task.

- We evaluate our approach on a real-world dataset. Experimental results show that our approach is feasible and effective. Moreover, our approach is very flexible and easy for future extension.

The rest of paper is organized as follows. Section II lists the related work. Section III introduces the approach details. Section IV shows the experiment results and conclude in Section V.

II. RELATED WORK

Our work is mainly related to three lines of research:

A. Developer recommendation

Recently, there are a considerable amount of research works about developer recommendation. Anvik et al. [1] applied a machine learning algorithm (Support Vector Machine) to the open bug repository to learn the kinds of reports each developer resolves for bug assignment to developers. Xuan et al. [23] combines naïve Bayes classifier and expectation-maximization (EM) to take advantage of both labeled and unlabeled bug reports for bug triage. Xie et al. [21] proposed DRETO. It models developers interest in and expertise on bug resolving activities based on topic models that are built from their historical bug resolving records. Given a new bug report, DRETO recommends a ranked list of developers who are potential to participate in and contribute to resolving the new bug according to these developers interest in and expertise on resolving it. Naguib et al. [13] proposed an approach for assignee recommendations using activity profiles. Yue Yu et al. [24] proposed a machine-learning based approach and a comment networks based approach of bug triaging for reviewer recommendation. Tao Zhang et al. [26] used topic model and developer relations to recommend the most suitable fixer for bug resolution. Wu et al. [19] proposed DREX to developer recommendation based on K-Nearest-Neighbor search with bug similarity and expertise ranking with various metrics, including simple frequency and social network metrics. Developer recommendation in software crowdsourcing is quite different from those in tradition applications. There are some more specific characteristics of crowdsourcing such as credit scoring. Traditional approaches can not leverage these kinds of information. Our approach can incorporate different kinds of information from crowdsourcing platform by leveraging a learning-to-rank framework. Moreover, we design a CRF-based approach to extract skill and location requirements from tasks’ description. Thus, our approach can make a more precise recommendation decision than traditional text similarity based approaches.

B. Crowdsourcing

There have been many research works about crowdsourcing including crowdsourcing applications [15], [17], [18], [25], quality-control techniques, pricing mechanisms in crowdsourcing [16], [14], etc. The studies most related to our work are task assignment techniques. Yudian Zheng et al. [29] proposed an optimization algorithm for Jury Selection Problem (JSP). Ju Fan et al. [7] introduced an adaptive crowdsourcing platform named iCrowd. It on-the-fly estimates accuracies of a worker by evaluating her performance on the completed tasks, and predicts which tasks the worker is well acquainted with. Yudian Zheng et al. [30] explored how Accuracy and F-score, two widely-used evaluation metrics for crowdsourcing applications, can facilitate task assignment, and proposed solutions that attain high quality in linear time for task assignment. Although existing methods perform well in simple crowdsourcing tasks, they may have limitations on more complicated software crowdsourcing tasks which require domain knowledge. The most recent research is [27]. They proposed a detailed task and developer model and used a regression-based model to construct the score function for recommendation in software crowdsourcing. However, the proposed task and developer model is too idealistic because most software crowdsourcing platforms cannot provide such well-structured information. Thus, it is impractical for real-world applications. In this paper, we propose a learning to rank based approach. Specifically, our approach leverages several features from different aspects including skill-matching features and topic-based features to measure the compatibility between tasks and developers. These features are learned from the description of tasks. Then a learning to rank algorithm is applied to rank the developers for each task.

C. Learning to Rank

Learning to rank is a type of supervised techniques whose goal is to automatically construct a ranking model from training data. It is first proposed in Information Retrieval (IR) area. Different from binary classification models, learning to rank aims to improve the ranking performance of the retrieved results [11]. In this paper, we formulate the developer recommendation task as a learning to rank problem and evaluate various learning to rank algorithms on the features derived from tasks and developers. The learning to rank algorithms we tested include pointwise approaches, MART [9] and Random Forests [3], pairwise approaches, RankNet [4] and RankBoost [8], and listwise approaches, ListNet [5] and AdaRank [22].
III. APPROACH

In this section, we start with a brief introduction of our proposed approach, and then describe it in details.

A. Approach Overview

We now provide a workflow to explain the whole process and how different components interact with each other. As shown in Fig. 1, we have three main components, namely CRF-based Criteria Extraction, Feature Engineering and Learning to rank. The input of CRF-based Criteria Extraction is a set of descriptions of tasks and developers collected from a software crowdsourcing platform. CRF-based Criteria Extraction tries to extract some criteria (i.e. skill and location) from those descriptions and learns the characteristics of tasks and developers. All the task-developer pairs are fed to Feature Engineering to extract features like skill-matching features and the topic-based features. A learning to rank algorithm is adapted to rank the developers of each task for developer recommendation.

B. Criteria Extraction from Task Description

In many crowdsourcing platforms, we have found that users might explicitly mention criteria information in their task descriptions which can not be directly used. For example, in the sentence “最好人在北京(Developers from Beijing are better)”, the user mentioned that the developers from Beijing are more suitable for this task. That is to say, “Beijing” is a location criteria. In the sentence “此项目需使用html5开发(This project should be developed using html5)”, the user means that the target developer should be familiar with “html5”. So “html5” is a skill criteria. These criteria information are very important to filter out those developers who cannot meet the requirements. If we treat the whole task description as text fragment, these kinds of information may be ignored due to the noise and sparseness.

In this section, we introduce our criteria extraction algorithm in detail. There are mainly four steps: choosing candidate sentences, labeling training samples, learning the criteria extraction model, and extracting criteria using the learned model. We model the criteria extraction from text sentences as a sequence-labeling problem and apply conditional random fields (CRF) [10] to train criteria extraction model. Note that we only extract location and skill information in this paper because these two types of information are very general and important in many software crowdsourcing platforms.

1) Sequence Labeling Problem: In machine learning, sequence labeling involves the algorithmic assignment of a categorical label for each member of a sequence of observed values. Input X is a sequence of observations and output Y represents hidden sequential states that need to be inferred from the observations; all the output y_i form a chain with an edge between each y_i-1 and y_i, which means that they follow the first-order Markov assumption. Commonly used sequence labeling models are the Hidden Markov Model (HMM) [6], Maximum Entropy Markov Model (MEMM) [12] and Conditional Random Field (CRF) [10]. CRF is a discriminative undirected probabilistic graphical model that is used to encode known relationships between observations and to construct consistent interpretations. It is often used for labeling sequential data, such as natural language text and its applications include word segmentation, part-of-speech tagging, named entity recognition, relation extraction and so on.

2) Candidate Selection: To extract criterias, the first step is selecting candidate sentences which may contain such criterias. In this paper, for location and skill extraction, we use two simple but effective strategies to select candidates.

- Candidates for location extraction We use a tagging based method to select candidates for location extraction. For each sentence, after doing word segmentation and named entity recognition (NER), those sentences which contain words tagged as location are considered as candidates for location criteria extraction. Here, we use Ansj\(^6\) for NER. The locations are tagged as ns or nss.
- Candidates for skill extraction We use a dictionary matching method to select candidates for skill extraction. A large scale software programming taxonomy has been built in our previous work [31]. It contains over 30,000 software programming terms\(^7\). Here, we just use these terms to compose our dictionary. For each sentence, after doing word segmentation, if it contain terms in the dictionary, then the sentence is considered as candidate for skill criteria extraction.

3) Modeling: In criteria extraction, the observable variables X, are word sequences and POS tag sequences and the hidden states Y, are tags defined by us that identify criteria.

\(^6\)https://github.com/NLPChina/ansj_seg
\(^7\)The data is available at http://datahub.io/dataset/software-zhishi-schema. We have also developed a website for better demonstration. http://zhishi.me/software
words in $X$; we assume $Y$ is satisfied with the first-order Markov assumption about dependencies among the output states. Therefore, we can model the task as a sequence-labeling problem. We adopt the CRF model, and train a CRF model using PocketCRF\(^8\) and name it Criteria Extraction-CRF (CE-CRF).

Given a candidate sentence, we label the criterias in the sentence as “C”. Other words which do not indicate the criterias are labeled as “O”.

![Figure 2: A labeled examples of CE-CRF](https://example.com/fig2.png)

4) **Feature Selection for Extraction**: In the extraction phase, the features at word level and POS level are adopted and we do not use any deep-NLP features. On the one hand, we argue that the two-level features are competent enough to train models for criteria extraction; on the other hand, the performance of deep NLP tools is doubtful, with most achieving a precision under 0.75. After experiment, we also find that basic unit features are sufficient to achieve satisfactory performance and that the over-inclusion of complex features can compromise the performance.

We use the unigram word (or POS) features and bigram word (or POS) features. The surrounding contextual words of criterion keywords are seen as potential evidence for criterion extraction. For example, a group of word level features we use is 

\[
W_{-2}^1, W_{-1}^1, W_0^1, W_{+1}^1, W_{+2}^1, W_{-1}W_0^1 \quad \text{and} \quad W_0W_{+1}^1,
\]

where $W$ stands for a word, index $0$ indicates the current word in focus and indices $-n$ to $n$ indicate the $n^{th}$ word to the left/right of the current word. Similarly, an example group of POS level features is

\[
\text{"POS}_{-2}^1", \text{"POS}_{-1}^1", \text{"POS}_0^1", \text{"POS}_{+1}^1", \text{"POS}_{+2}^1", \text{"POS}_{-1}P_{+1}^1 \quad \text{and} \quad \text{"POS}_0P_{+1}^1".
\]

Intuitively, the word level features will lead to good precision and the POS level features will raise the recall. Fig. 2 shows a labeled training samples. The first layer is the labeled results. The second layer uses the POS level features while the third layer uses the word level features.

For input sentences, by initially performing word segmentation and POS tagging, those candidate sentences are recognized by the method described in Section III-B2. Then, CE-CRF makes a single pass over them and labels each word. The result of the labeling indicate whether a word is a criteria keyword. Note that our criteria extraction algorithm is quite general. The extraction results just depend on the labeled data.

---

\(^{8}\)http://code.google.com/p/pocketcrf/

---

C. **Feature Engineering**

The purpose of feature engineering is to quantitatively characterize the similarities or relatedness between tasks and developers. We define several features to characterize the task-to-developer compatibility. The details of these features are as follows:

**Task-independent Features**: This type of features is independent of a specific task. Intuitively, if a developer is more successful with higher rating scores, the crowdsourcer is more likely to choose him/her. Based on this intuition, we design features based on the rating scores. However, it is not suitable to treat the rating scores as features directly. The level of a developer depends not just on the rating scores but also on the confidence of the rating scores, while the confidence of the rating scores depends on the number of people who score the developer. For example, a developer $A$ has finished 1 task and his score is 5 while another developer $B$ has finished 100 tasks and his score is 4.9. It is obviously that $B$ is more successful than $A$ because $B$ is more experienced and the rating score is also high. However, if we directly use the rating score as features, we will get a completely opposite ranking result.

In this paper, we apply Wilson interval\(^9\) to normalize the rating scores. Wilson interval has good properties even for a small number of trials and it is widely used in statistic analysis. Given the original rating score and the number of people who have participated the scoring process, we will get a confidence interval and the lower bound of the interval will be the normalized rating score. Here we set the parameter $a$ of Wilson interval to 5\(^{10}\). Back to the example we mentioned above, after applying Wilson interval, the score of developer $A$ is 1.03 while the score of developer $B$ is 4.65, which indicates that developer $B$ is more successful.

Note that there may be many types of scores in a software crowdsourcing platform. For example, in Zhubajie, the largest software crowdsourcing platform in China, there are six types of scores, namely favourable ratio, reputation score, service related score, efficiency related score, quality related score and overall score. Topcoder uses different types of badges to measure the qualifications of developers. It is very easy to map these badges to scores. Thus, all you need to do is to normalize these scores using Wilson interval and feed them into the learning process. So our approach is quite flexible.

**Task-dependent Features**:

1) **Topic-based Task-Developer Similarity**: This feature focused on making independent developer recommendation decisions using the local task-user compatibility. The essential idea is to recommend developer by comparing the contextual similarity with each of its candidate developers. Instead of using a TFIDF-based method, we try to represent each task and developer in a semantic level and design a topic-based similarity as a feature. We leverage a topic modeling method named Latent Dirichlet Allocation (LDA) [2] to generate a topic based representation for each task and developer. LDA is a statistical technique that expresses each document as a probability distribution of topics. Each topic is itself a probability distribution of words referred to as a topic model. Words can

\(^{9}\)https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval

\(^{10}\)This means the confidence level is 95%.
belong to multiple topics, and documents can contain multiple topics. In our proposed method, we model task and developer by its description, and learn its topic representation by LDA. In our experiments, the number of topics was empirically set as T = 50 and the hyper parameters α and β were set with α = 50/T and β = 0.01 respectively. We ran Gibbs sampling for 1,000 iterations.

Let P(t) be the topic distribution of the task t, and P(d) be the topic distribution of the developer d. There are several ways one can compare these probability distributions to estimate the semantic similarity of the t and d. One standard way of doing so is to use the Kullback-Leibler (KL) divergence: given two probability distributions P and Q, their KL divergence measures their distance, as follows:

\[ D_{KL}(P||Q) = \sum_{i=1}^{K} P_i \log \frac{P_i}{Q_i} \]  

(1)

Note that it is an asymmetric metric, i.e. \( D_{KL}(P||Q) \neq D_{KL}(Q||P) \). In this work, we use Jensen-Shannon divergence. It is based on the KL divergence, with some notable differences, including that it is symmetric and it is always a finite value.

\[ JSD(P||Q) = \frac{1}{2} D_{KL}(P||M) + \frac{1}{2} D_{KL}(Q||M) \]  

(2)

where \( M = \frac{1}{2}(P + Q) \).

Thus, the topic-based task-developer similarity between a task t and a developer d is compute as:

\[ Sim_{topic}(t, d) = JSD(P(t)||P(d)) \]  

(3)

2) Topic-based Task-Task Similarity: Similarly, we also use the topic model to compute the similarity between given task t and the historical tasks of a candidate developer. Since there may be several historical tasks for a developer, so given a task t and a set of historical tasks \( T_d \) of developer d, here, we use the average value and the maximum value of the topic-based similarities as two features.

\[ Avg_{topic}(t, d) = \frac{\sum_{t_i \in T_d} JSD(P(t)||P(t_i))}{|T_d|} \]  

(4)

\[ Max_{topic}(t, d) = \max_{t_i \in T_d} JSD(P(t)||P(t_i)) \]  

(5)

3) Skill Matching: The skill matching feature aims to quantitatively characterize the similarity or relatedness between the task and developer in term of skill. We now propose a semantic-based skill matching algorithm. The detail is as follows:

Given a task t, we can extract its required skills \( S_t \) using our extraction algorithm in Section III-B. Similarly, the skills of a developer d can also be obtained from his historical tasks. Here, we do not directly extract skills from the historical tasks and treat all skills equally. Instead, we use probabilities to characterize the skills of developers. Specifically, We maintain a counter \#s which counts the number of times the skill occurred in the historical tasks. Finally, we use Laplace smoothing (a.k.a add-one smoothing) to estimate the skill distribution of a developer d as follows

\[ p_s^{(d)} = \frac{\#s + 1}{\sum_{t \in S} \#s' + |S|} \]  

(6)

where S is the set of all skills of developer d. \( p_s^{(d)} \) is a valid probability since \( \sum_{s \in S} p_s^{(d)} = 1 \). Thus, we can model the skills as a probability distribution, and a higher probability value of a skill s indicates that the developer is more professional with that skill.

Given the required skill set \( S_t \) of a task t and the skill distribution \( p_s^{(d)} \) of a developer d. First, for each skill \( s_i \) in \( S_t \), we can compute its best matching score \( score_{s_i} \) in \( p_s^{(d)} \). The matching score is the combination of two parts. The first is the similarity between \( s_i \) and a skill \( s_j \) in \( p_s^{(d)} \) and the second is the degree of expertise on the skill \( s_j \).

\[ score_{skill}(s_i, d) = \max_{s_j \in p_s^{(d)}} sim(s_j, s) * p_s^{(d)} \]  

(7)

The intuition behind this idea is that even though there exists a skill \( s_j \) in \( p_s^{(d)} \) which holds a high semantic relatedness with \( s_i \), but the developer is actually not professional with \( s_j \). That means the value of \( p_s^{(d)} \) is rather low. In this circumstance, \( s_i \) is not a good matching result of \( s_i \).

Here, the \( sim(.) \) function is used to compute the semantic similarity of two software development skills such as “java” and “python”. It is difficult to compute the term similarity using traditional methods such as knowledge-based similarity computing methods or statistical-based similarity computing methods. The reason comes from two aspects. First, traditional knowledge base such as WordNet or Wikipedia are very general, some domain-specific terms may be missed in them and the information from a general knowledge base is a bit noisy which will reduce the performance of similarity computing. Second, the statistical-based methods need a large-scale training corpus to train the statistic model which is sometimes impossible for software engineering research. Here, we leverage our previous work and design a tree-based similarity computing method on our software programming taxonomy\textsuperscript{11} [31]. Experimental results have shown that compared with traditional methods such WordNet-based method, ESA-based method and Web-based method, our taxonomy can significantly boost the performance of term similarity computing\textsuperscript{12}. The computation of the term similarity on a tree is actually WUP similarity [20]. The detail is as follows:

\[ sim(s_1, s_2) = \frac{2 \times \text{depth}(LCA(s_1, s_2))}{\text{depth}(s_1) + \text{depth}(s_2)} \]  

(8)

Where \( LCA \) denotes the lowest common ancestor of \( s_1 \) and \( s_2 \) in our taxonomy.

Base on the best matching score, given a required skill set \( S_t \), we use the average value and the maximum value of the best matching scores as two features.

\[ Avg_{skill}(t, d) = \frac{\sum_{s_i \in S_t} score_{skill}(s_i, d)}{|S_t|} \]  

(9)

\textsuperscript{11}We also use this taxonomy to compose a term dictionary in Section III-B2.

\textsuperscript{12}The experiment is out the scope of this paper and the details can be found at http://seonto.apexlab.org/termsim.
\[
Max_{skill}(t, d) = \max_{s_i \in S_t} score_{skill}(s_i, d)
\] (10)

4) Constraint Filtering: As we have mentioned above, the location criteria in the task description is very important because the crowdsourcers may need a face-to-face communication with the developers. So we use our CE-CRF model to extract location criteria in the task description. For developers, it is easy to obtain their profiles since most crowdsourcing platforms require developers to fill in some attributes in his/her public profile during registration, which can be modified later.

Here, we design a binary feature to represent this kind of constraints, if the location information in the developer’s profile is the same as the location requirement in the task description, then the value of this feature is 1. Otherwise, the value will be 0. Note that many constraints can be represented as this kind of features such as bid constraint, work experience constraint or even sex constraint. We can see that it is easy to add these constraints to our approach.

D. Learning to Rank for Developer Recommendation

Now we discuss how to devise the developer recommendation algorithm. We develop a learning to rank framework for developer recommendation and the framework is quite flexible that can incorporate various types of useful information. Moreover, our proposed approach is generic that any learning to rank algorithms can be easily applied. Although the binary classification approach is a natural and simple way to deal with the developer recommendation task [24], it has several drawbacks. Firstly, the training data is very unbalanced since only one developer could be selected and the vast majority of candidate developers are negative examples. Moreover, when multiple candidate developers for a task are classified as positive by the binary classifier, they have to utilize other techniques to select the most likely one.

Learning to rank is a type of supervised techniques whose goal is to automatically construct a ranking model from training data. Training data for the learning to rank model consists of lists of items with some partial order specified between items in each list. While for the problem of developer recommendation, the approaches just focus on the single best developer in the candidate developer set and therefore impose a loose requirement, that is, the most suitable developer should be ranked highest. This formulation addresses the problems with the binary classification. Firstly, training data is balanced since we have a single ranking example for each task. Secondly, tasks just need to select the candidate developer which achieves the highest score in the test phase, rather than resorting to other techniques to select the most likely one.

There are three types of learning to rank algorithms [11], [28]:

- **Pointwise approach.** It reduces ranking to regression or classification on single documents. That is, we build the ranking function \( f \) by approximating \( f(x^{(i)}_j) \) to the corresponding relevance score \( y^{(i)}_j \).
- **Pairwise approach.** It no longer assumes absolute relevance, which reduces ranking to classification on developer pairs w.r.t. the same query. That is, given a task \( i \), we only focus on the relative preference order between a pair of candidate developers \( d^{(i)}_j \) and \( d^{(i)}_k \).
- **Listwise approach.** Instead of reducing ranking to regression or classification, it performs learning directly on developer lists, and an entire ranked list is treated as a learning instance. That is, for task \( i \), the ranking function focuses on learning a list of relevance scores \( (y^{(i)}_1, ..., y^{(i)}_n) \) given the list of feature vectors \( (x^{(i)}_1, ..., x^{(i)}_n) \).

We define our developer recommendation problem in a generic way such that all the above approaches can be used.

IV. EXPERIMENTS

In this section, we show the experimental results of our proposed method.

A. Experiment Setup

Specifically, we focus on evaluation of criteria extraction and developer recommendation.

1) Data Statistics: In order to evaluate our approach, we crawl data from Zhubajie.com\(^\text{13}\) which is the largest software crowdsourcing platform in China. There are many task categories in Zhubajie, only software development related categories (i.e. website development, desktop and mobile application development) are considered in this paper. In total, there are 2,219 tasks and 1,487 developers. All the data used in our experiments is available at https://github.com/zhujiangang/SoftwareCrowdsourcing.

B. Evaluation on Criteria Extraction

1) Construction of the dataset.: To construct the dataset for the evaluation of criteria extraction, we first randomly select 1,000 sentences which contain at least one keyword expressing skill or location. Then three students are asked to label the sentences whether they express the criteria. By only keeping the sentences which receive the same labels from all the annotators, we end up with 751 criteria sentences and 249 non-criteria sentences.

2) Experimental Metrics: We use precision, recall, and F1-Measure to measure the performance of our proposed extraction approach, which are defined as follows:

**Precision \( p \):** It is the percentage of correctly discovered criteria in all discovered criteria.

\[
p = \frac{|A \cap T|}{|A|}
\] (11)

where \( A \) is the set of discovered criteria, and \( T \) is the set of ground truth criteria.

**Recall \( r \):** It is the percentage of correctly discovered criteria in all correct criteria.

\[
r = \frac{|A \cap T|}{|T|}
\] (12)

\(^\text{13}\)http://www.zhubajie.com/
TABLE I: The performance comparison of criteria extraction

<table>
<thead>
<tr>
<th>Methods</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary-based</td>
<td>0.581</td>
<td>1.000</td>
<td>0.735</td>
</tr>
<tr>
<td>Rule-based</td>
<td>0.966</td>
<td>0.602</td>
<td>0.742</td>
</tr>
<tr>
<td>Ours</td>
<td>0.892</td>
<td>0.833</td>
<td>0.861</td>
</tr>
</tbody>
</table>

**F1-Measure** (F1): F1-Measure (also known as F1 score) considers the overall result of precision and recall and is the harmonic mean of precision and recall.

\[ F1 = \frac{2pr}{p + r} \]  

(13)

3) **Comparison Methods:**

- **Dictionary Matching** This method is just the same as our candidate selection method in Section III-B2. All the keywords which can be matched in the dictionary are considered as criterias.

- **Rule-based Extraction** This method applies some linguistic rules for extraction. A typical rule for location extraction is “Developers from X are preferred”. Here, “X” is considered as a location criteria.

4) **Result Analysis:** It can be observed from the results presented in Table I that using our proposed CE-CRF outperforms the baselines. In particular, the dictionary based method achieves the recall of 1 while the precision of it is rather low. This is obvious because all the possible criteria keywords must be in the dictionary while not every keyword which can found in the dictionary is criteria due to the semantic noise. The rule base method achieves a high precision but its recall is low. This is mainly because the natural language expressions of criteria are various. Our method performs well both in precision and recall, and thus achieves the highest f1-score.

C. Evaluation on Developer Recommendation

1) **Evaluation Metric:** The evaluation metrics adopted here for evaluation on developer recommendation is precision at k (p@k), which is commonly used in information retrieval. To compute p@k, we calculate the proportion of tasks for which we have made the correct recommendations in the top k positions.

2) **Comparison Methods:** We evaluate our proposed developer recommendation system using two representative learning to rank algorithms from each of the categories, pointwise, pairwise and listwise:

- **Pointwise:** MART [9], Random Forests (RF) [3];
- **Pairwise:** RankNet [4], RankBoost [8];
- **Listwise:** ListNet [5], AdaRank [22].

3) **Feature Contribution Analysis:** How much does each feature contribute to developer recommendation? In order to get some insight to this question, we perform an analysis to evaluate the contribution of each feature. Here, we use the real-world data crawled from Zhubajie as the ground truth. Then we train five rankers, and each ranker removes one feature from the whole feature set (T denotes the topic-based features, SM denotes the skill-matching features and CF denotes the constraint-filtering feature). We apply 5-fold cross validation to train the five rankers. Fig. 3 shows the impact of each feature on the performance. According to the decrement of P@1, all these features are useful in developer recommendation. The most effective features are the Topic-based Features. The performance is degraded sharply. This indicates that the description and the historical tasks of the developers are very important in software crowdsourcing. Task-independent features are also important for developer recommendation. This follows the intuition that “rich-gets-richer”. If a developer is more successful with higher rating scores, the task owner is more likely to choose him/her. Although the feature of constraint filtering seems to be less important than others, we find it is indeed helpful to improve the performance.

4) **Overall Performance Comparison:** It can be observed from Table II that all the learning to rank methods perform well in our dataset. Furthermore, the performance of the pointwise approach is better than that of pairwise and listwise approaches. This is mainly because it is difficult to obtain the full developer ordered list for training since there is usually only one correct developer decision and the ordering of irrelevant developers is not available. In the pointwise category, MART works very well and nearly achieves the best performance for p@k.

V. CONCLUSION AND FUTURE WORK

In this work, we propose a learning to rank based approach for effective developer recommendation in software crowdsourcing. Specifically, our approach leverages several features from different aspects including skill-matching features and topic-based features to measure the compatibility between tasks and developers. These features are learned from the...
description of tasks. Then a learning to rank algorithm is applied to rank the developers for each task. The experiments on real-world dataset show the feasibility and effectiveness of our approach.

As for future work, we will try to evaluate our approach on more real-world datasets such as Topcoder. Moreover, it would be interesting to explore the efficiency of our approach for real-time recommendation.

VI. ACKNOWLEDGMENT

This research is supported by 973 Program in China (Grant No. 2015CB352203) and National Natural Science Foundation of China (Grant No. 61472242).

REFERENCES


