A Top-*k* Learning to Rank Approach to Cross-Project Software Defect Prediction

Feng Wang
School of Computer Science
Wuhan University
Wuhan 430072, China
waple0820@gmail.com

Jinxiao Huang
School of Computer Science
Wuhan University
Wuhan 430072, China
mia107@whu.edu.cn

Yutao Ma
School of Computer Science
Wuhan University
Wuhan 430072, China
ytma@whu.edu.cn

***Abstract*—Cross-project defect prediction (CPDP) has recently attracted increasing attention in the field of Software Engineering. Most of the previous studies, which treated it as a binary classification problem or a regression problem, are not practical for software testing activities. To provide developers with a more valuable ranking of the most severe entities (e.g., classes and modules), in this paper, we propose a top-*k* learning to rank (LTR) approach in the scenario of CPDP. In particular, we first convert the number of defects into graded relevance to a specific query according to the three-sigma rule; then, we put forward a new data resampling method called SMOTE-PENN to tackle the imbalanced data problem. An empirical study on the PROMISE dataset shows that SMOTE-PENN outperforms the other six competitive resampling algorithms and RankNet performs the best for the proposed approach framework. Thus, our work could lay a foundation for efficient search engines for top-ranked defective entities in real software testing activities without local historical data for a target project.**

Keywords*—top-k ranking, relevance, resampling, Wilcoxon signed-rank test, transfer learning

I. Introduction

In the past decade, software defect prediction (SDP) has played an increasingly important role in software quality assurance and attracted widespread attention from the field of Software Engineering. Up to now, a large number of methods have been proposed in previous studies to predict the defect-proneness of software entities such as object-oriented classes and code modules. Generally speaking, these SDP methods belong to two categories: within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) [1]-[8]. In the scenario of WPDP, training data and test data are from the same software project. Unlike WPDP, in the scenario of CPDP training data is collected from other favorite projects or publicly-available datasets, mainly due to a lack of historical data (or labeled data) in newly-created or inactive projects [9]. Besides, considering the popularity and efficacy of transfer learning (or called inductive transfer) [10] in a few practical applications, the CPDP approach using transfer learning has drawn much attention recently [6]-[8], [11]-[13].

To the best of our knowledge, in both the scenarios of WPDP and CPDP, an overwhelming majority of previous works considered SDP as a binary classification problem [1]-[9], [11]-[13]. For a given software entity, a binary classifier, also known as a (binary) classification model, can predict whether it is defective or buggy. However, this type of approaches based on binary classification is not practical for real-world software testing processes that have limited human resources and tight deadlines [14]. Hence, some researchers attempted to revisit SDP as a regression problem in the scenario of CPDP [14]-[20]. After a regression model predicts how many defects or bugs are involved in each software entity for testing, a software tester will obtain a ranking of defective entities sorted by their ranks or scores which are proportional to the number of defects. Moreover, different types of multiple regression models with optimization algorithms have been put forward to predict the number of each software entity's defects as accurately as possible [15]-[17].

Even so, it is worth noting that accurate rank estimations at the individual entity level may not necessarily lead to better sorting results. Here, we illustrate a simple example in Fig. 1. In Fig. 1, there are four classes denoted as *A*, *B*, *C*, and *D*, respectively. *A*, *B*, and *C* have four bugs, three bugs, and one bug, respectively, while *D* is not defective. Suppose that there are two trained regression models denoted as M_1 and M_2, respectively. Fig. 1 shows that M_1 performs better than M_2 regarding commonly-used evaluation metrics such as root mean square error (RMSE) and mean absolute error (MAE). However, according to the result of M_1, *B* has a higher rank than *A*. On the contrary, the result of M_2 shows a more accurate ranking of the four classes. Thus, we argue that recommending satisfactory rankings is probably more valuable to practical software testing activities than the regression-based methods themselves. In other words, we indeed need an automatic machine-learned ranking approach to produce new (ordered) lists of defective software entities in a way similar to rankings in training data in some sense.

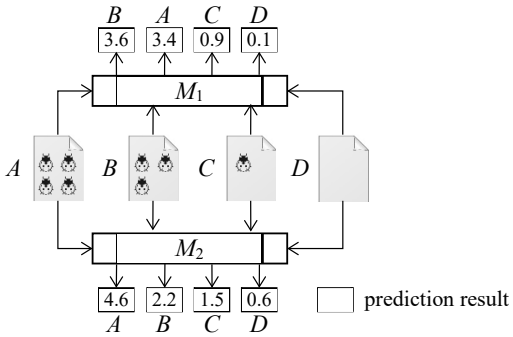

Fig. 1. An illustration of a simple example showing the necessity of an accurate sorting result for four classes regarding the number of defects.

Learning to Rank (LTR), known as one of the essential approaches in the field of Information Retrieval (IR) [21], is now developing rapidly to construct ranking models for a few application areas, including IR, recommender systems, and machine translation. However, there is very little research on the application of this machine learning method to SDP. On the one hand, few of researchers in the Software Engineering field regarded SDP as a ranking problem, or more specifically, the top-*k* ranking problem [22], although we acknowledge that ranking is one of the fundamental questions for the entire field of Computer Science. On the other hand, there is, of course, a

978-1-7281-1970-0/18/$31.00 ©2018 IEEE
DOI 10.1109/APSEC.2018.00048

deficiency in freely-available bug datasets including queries and documents matching them together with each match's relevance degree. Note that any software entity can be deemed to be a type of documents in this study.

Despite the promising results inspired by the Pointwise approach to LTR, which is almost approximated by regression methods [15]-[17], there is still much room for improvement in prediction performance, especially for rankings of the top-ranked defective software entities. Considering the priority of fixing software defects, in this work, we will focus on how to predict an ordered list of the top *k* buggy entities with LTR as accurately as possible. Besides, a comprehensive comparison of eighty-seven commonly-used LTR methods [23] shows that, on the whole, the Listwise approach performs better than the Pairwise and Pointwise approaches. Thus, the goal of this study is to propose a top-*k* LTR approach with the Listwise approach or Pairwise approach to construct ranking models in the scenario of CPDP. Also, we will empirically validate the feasibility of the proposed approach in an experimental dataset transformed from the widely-used PROMISE dataset [24]. In short, the main contributions of this work are two-fold.

1. In the scenario of CPDP, we first propose an LTR approach that learns from imbalanced data to predict a ranking of the top *k* defective software entities. The framework of our method has three core components, namely relevance calculator, data resampler, and LTR algorithm selector, and it can work in a manner that returns those documents which are more relevant to a given query.
2. By using the relevance calculator and data resampler of the proposed approach, we obtain a new dataset including query-document pairs transformed from the widely-used PROMISE dataset in the SDP field. This dataset could be further employed to be a benchmark dataset for comparing different LTR approaches to SDP, such as RankNet [25] and ListNet [26], in the scenario of CPDP.

The rest of this paper is organized as follows. Section II reviews the related work of CPDP using regression methods. Section III introduces the framework of our method. Sections IV and V present the experimental design, setups, and results. Section VI discusses some issues related to our approach and potential threats to the validity of our work. Finally, Section VII concludes this work and presents our future work.

II. Related Work

In this section, we review the related work on predicting the number of software defects and the ranks of software entities. Note that we will not discuss the CPDP approaches based on binary classification. If readers are interested in them, please refer to some recent surveys about CPDP [6]-[8], [11]-[13].

A. Regression-Based Prediction Approaches

Previous studies usually trained defect prediction models (or called predictors) using statistical methods (such as linear regression [27] and multiple regression analysis [28]) and employed them to predict the number of defects in a given software entity. Because these methods were easy to use and efficient, they were then applied in the scenario of CPDP. For example, Chen *et al.* [14] trained six defect predictors using commonly-used regression methods to predict defect number. In both the two CPDP scenarios (i.e., *many-to-one* and *one-to-one*), the predictor built with a decision regression tree was the best estimator among the six predictors. Note that the *many-to-one* CPDP scenario indicates that for a given test set, there are several training data sets collected from other available projects. Also, a more detailed comparison of regression-based prediction approaches made by Rathore *et al.*, please refer to [18].

By using resampling and ensemble learning techniques, Yu *et al.* [20] recently attempted to tackle the imbalanced data problem in predicting the number of bugs. Similarly, Rathore *et al.* [19] presented a system built based on the heterogeneous ensemble method for the prediction of the number of software defects. Besides, Yu *et al.* [29] proposed a feature selection method using Feature Spectral Clustering and feature Ranking (FSCR) to predict the number of software faults. Moreover, this method was proved to be able to eliminate the effect of irrelevant and redundant features on the overall performance of prediction models.

B. Ranking-Oriented Prediction Approaches

Nguyen *et al.* [30] put forward a rank-based method to predict the ranks of software modules using machine learning, and the results of their empirical evaluation showed that this approach was credible in SDP. Another similar approach proposed by Yang *et al.* [17] introduced an LTR approach to constructing defect prediction models for the ranking task. In particular, they directly optimized sorting results measured in terms of the metric *FPA* (Fault Percentile Average), using the method of composite differential evolution. However, both the two studies were conducted only in the scenario of WPDP. That is to say, the external validity and applicability of the two approaches mentioned above in the scenario of CPDP remain unknown. Inspired by the Pointwise approach, You *et al.* [15], [16] put forward an approach called ROCPDP (Ranking-Oriented Cross-Project Defect Prediction) and conducted an empirical study in two specific CPDP scenarios, namely *many-to-one* and *one-to-one*. Besides, the high prediction performance of the proposed approach in the two datasets of AEEEM [31] and PROMISE [24] validated its effectiveness.

In general, since these approaches outputs rankings of defective software entities using various regression methods, none of them falls within the scope of the query-level LTR approach. Hence, this work focuses on the LTR approach for query-focused multi-document prediction, which allows us to run a query for matching relevant documents and then return the top *k* documents.

III. The Framework of Our Approach

As stated earlier, our approach has three core components, i.e., relevance calculator, data resampler, and LTR algorithm selector. Unlike the approaches presented in Subsection II.B, which are roughly approximated by a regression technique, our approach is a top-*k* ranking approach using LTR in the real sense.

A. Relevance Calculator

In the field of IR, a search engine always ranks candidate documents according to their relevance to a given query. Unlike those previous studies using the defect-proneness or the number of defects as class labels, this work focuses on relevance judgments of documents to queries. For simplicity,

a simple query is defined as "*Is the given software entity severe?*" We then calculate the relevance of each entity to this query based on the number of defects.

Given a set of software entities $E = (e_1, e_2, \cdots, e_n)^T$, each entity e_i has x_i ($x_i \in \mathbb{N}$) software defects. Since the variable X, the number of each entity's defects, is a natural number, we can fit the distribution of defects using the probability density function of the folded normal distribution [32]. Compared with exponential distributions and power-law distributions, such a distribution can convert natural numbers into different levels by using the well-known three-sigma rule of thumb. Given a sample x_i of size n, the log-likelihood of the folded normal can be formulated as follows [33].

$$l = -\frac{n}{2}\log 2\pi\sigma^2 - \sum_{i=1}^{n}\frac{(x_i-\mu)^2}{2\sigma^2} + \sum_{i=1}^{n}\log\left(1 + e^{-\frac{2\mu x_i}{\sigma^2}}\right), \quad (1)$$

where μ and σ^2 are the mean and variance of the variable X.

The partial derivatives of the log-likelihood concerning μ and σ^2 are written as

$$\frac{\partial l}{\partial \mu} = \frac{\sum_{i=1}^{n}(x_i-\mu)}{\sigma^2} - \frac{2}{\sigma^2}\sum_{i=1}^{n}\frac{x_i}{1+e^{\frac{2\mu x_i}{\sigma^2}}}, \quad (2)$$

$$\frac{\partial l}{\partial \sigma^2} = -\frac{n}{2\sigma^2} + \frac{\sum_{i=1}^{n}(x_i-\mu)^2}{2\sigma^4} + \frac{2\mu}{\sigma^4}\sum_{i=1}^{n}\frac{x_i}{1+e^{\frac{2\mu x_i}{\sigma^2}}}. \quad (3)$$

Let the first partial derivative of the above log-likelihood (see (2)) be zero, we then obtain

$$\sum_{i=1}^{n}\frac{x_i}{1+e^{\frac{2\mu x_i}{\sigma^2}}} = \frac{\sum_{i=1}^{n}(x_i-\mu)}{2}. \quad (4)$$

By substituting (4) to (3) and equating it to zero, we will get the following expression for the variance.

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n} + \frac{2\mu\sum_{i=1}^{n}(x_i-\mu)}{n} = \frac{\sum_{i=1}^{n}(x_i^2-\mu^2)}{n} = \frac{\sum_{i=1}^{n}x_i^2}{n} - \mu^2. \quad (5)$$

(4) and (5) can be used to obtain maximum likelihood estimates in a common recursive way. Recently, Tsagris *et al.* [34] suggest a more natural and more efficient way to perform a search algorithm for the mean and variance. Then, (4) can be rewritten in a more elegant way as

$$2\sum_{i=1}^{n}\frac{x_i}{1+e^{\frac{2\mu x_i}{\sigma^2}}} - \sum_{i=1}^{n}\frac{x_i\left(1+e^{\frac{2\mu x_i}{\sigma^2}}\right)}{1+e^{\frac{2\mu x_i}{\sigma^2}}} + n\mu = 0, \quad (6)$$

i.e.,

$$\sum_{i=1}^{n}\frac{x_i\left(1-e^{\frac{2\mu x_i}{\sigma^2}}\right)}{1+e^{\frac{2\mu x_i}{\sigma^2}}} + n\mu = 0, \quad (7)$$

where σ^2 is defined in (5). Normally, we can perform a maximization procedure using an expectation–maximization (EM) algorithm [35] until the change in the log-likelihood value for unknown parameters μ and σ^2 is negligible.

After obtaining the approximate estimates of the mean and variance (denoted as $\hat{\mu}$ and $\hat{\sigma}^2$ respectively) for a given data set, we can define some simple mapping rules to calculate each entity's relevance r_i to the query regarding the number of defects. According to the three-sigma rule, the probability of a sample falling outside 3σ is less than 0.3% [36]. Then, the rules for relevance calculation based on the three-sigma rule are formulated as

$$r_i = \begin{cases} 0, \text{if } x_i = 0 \\ 1, \text{else if } x_i \in (0, \hat{\mu}+\hat{\sigma}] \\ 2, \text{else if } x_i \in (\hat{\mu}+\hat{\sigma}, \hat{\mu}+2\hat{\sigma}] \\ 3, \text{else if } x_i \in (\hat{\mu}+2\hat{\sigma}, +\infty] \end{cases} \quad (8)$$

where the three numbers (i.e., "3," "2," and "1") represent the scale of grades. For example, "3" and "1" indicate high grade and low grade, respectively. Besides, we set the relevance value of any non-defective class file to "0."

Assuming that we get $\hat{\mu} = 0.5$ and $\hat{\sigma} = 3$ for a given dataset, if a class file contains more than seven defects, the relevance value of the class is set to "3," according to the last rule presented in (8). For buggy class files that contain four or five bugs, their relevance values are set to "2." Similarly, those class files which contain few bugs have a lower relevance to the query, and their values of the relevance to the query are set to "1."

B. Data Resampler

Because in a real-world data set only a small number of software entities are defective, SDP has also been considered to be an imbalanced (data) learning problem, especially for binary classification [37], [38]. According to the promising results of many previous studies [20], [39], data resampling is often a standard technique to tackle the imbalanced learning problem resulting in a decrease in classification performance. In this study, we also find that most of the software entities in our dataset have low relevance to the query. Considering that the main objective of this study is to recommend an accurate ranking of the top *k* high-risk samples that have high relevance values, we present a data resampler, called SMOTE-PENN (see Algorithm 1), for the dataset used in this study to address the imbalanced learning problem.

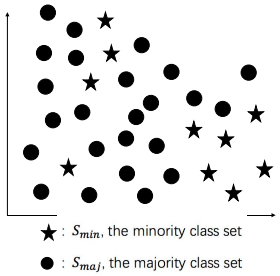

Fig. 2. An example of an imbalanced dataset.

In particular, the data resampler is an improved algorithm that combines an under-sampling technique entitled Edited Nearest Neighbor (ENN) [40] and an oversampling technique

called Synthetic Minority Oversampling Technique (SMOTE) [41]. For an imbalanced dataset, Algorithm 1 first divides the dataset into two subsets, S_{maj} and S_{min}, which contain the majority of normal samples (i.e., $r_i = 0$) and the minority of defective samples (i.e., $r_i \neq 0$), respectively, as shown in Fig. 2. For each sample $x_i \in S_{min}$, we identify and select its k_{min} nearest neighbors $NN(x_i)$ regarding the Euclidean distance. The algorithm then counts the number of majority classes in set $NN(x_i)$ and the number of the remaining minority classes (see the fifth line in Algorithm 1).

Inspired by the idea of the borderline-SMOTE [42], we categorize the minority of defective samples into "Sparse," "Isolated," and "Dense." As depicted in Fig. 3, an *isolated* sample is a buggy sample whose nearest neighbors are defect-free. In other words, it is difficult for these *isolated* samples to generate new defective ones around them. For a *sparse* sample, its nearest neighbors consist of a large number of normal samples and a small number of defective ones. On the contrary, most of the nearest neighbors of a *dense* sample are defective samples, and they tend to gather into a cluster (see the example in Fig. 3). Generating more *dense* samples may result in the model overfitting problem.

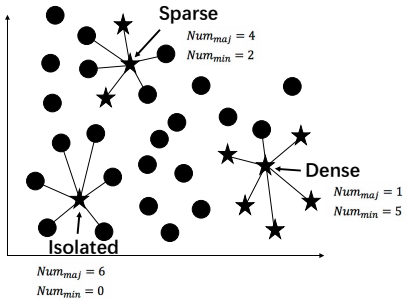

Fig. 3. Three examples of *isolated*, *sparse*, and *dense* samples.

For each sample in the set of *sparse* samples (see the sixth line in Algorithm 1), SMOTE-PENN randomly chooses one of its nearest neighbors e_j and then produce a synthetic defective sample at a random point between e_j and the sample in each iteration. Fig. 4 depicts such an iteration process. Here, L is a parameter used to ensure that the size of normal samples is larger than that of defective ones since Algorithm 1 generates a new defective sample for each *sparse* sample in an iteration. Therefore, the higher value of L, the larger size of the minority class set, thus increasing the proportion of defective samples to normal ones.

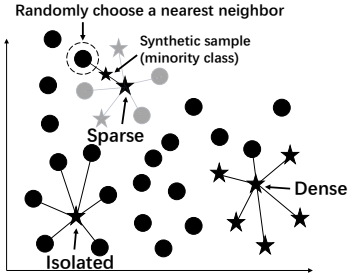

Fig. 4. An iteration for processing a *sparse* sample.

Then, this algorithm obtains the union set of the nearest normal neighbors of each defective sample in S_{min}, denoted as S_{bmaj}. Similarly, we identify the nearest k_{maj} neighbors $NN(y_i)$ for each normal sample y_i in S_{bmaj} and compute the proportion of normal samples. If the proportion is smaller than a threshold N (see the nineteenth line in Algorithm 1), the sample will be removed from S_{newmaj}. A simple example is referred to Fig. 5.

Here, M and N represent two thresholds used to determine the type of a sample. Take M as an example. Given a defective sample that has five neighbors (including two normal samples and three defective samples), it will be marked as "Sparse" if $M = 0.8$. We then add a new neighbor to the sample. If $M = 0.5$, the sample will be marked as "Dense" since the ratio of defective samples (i.e., 0.6) is greater than M. In general, the higher the value of M, the more samples will be added to S_{newmin} in Algorithm 1. Instead, if we increase the value of N, more samples will be removed from S_{newmaj} in Algorithm 1. In our experiment, we intuitively set $M = N = 0.5$ and let them remain unchanged.

After Algorithm 1 ends, we finally get a new training set whose proportion of defective samples to normal ones is kept within a reasonable range to tackle the problem of imbalanced data in the SDP field.

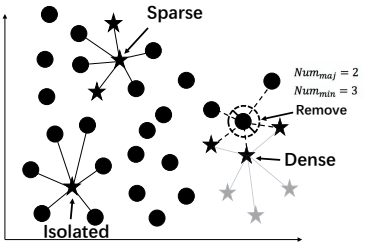

Fig. 5. The removal process of a redundant normal sample.

Algorithm 1: SMOTE-PENN

Inputs: S_{maj}, S_{min}, k_{maj}, k_{min}, M, N
Outputs: S_{newmaj}, S_{newmin}

01. Initialize $S_{newmaj} = S_{maj}$, $S_{newmin} = S_{min}$, $L = \left\lfloor \frac{|S_{maj}|}{|S_{min}|} \right\rfloor$;
02. **for** l = 1 to L **do**
03. **for** each $x_i \in S_{min}$ **do**
04. Compute k_{min} nearest neighbors $NN(x_i)$ by Euclidean distance;
05. Divide $NN(x_i)$ to $NN_{maj}(x_i)$ and $NN_{min}(x_i)$;
06. **if** $NN_{min}(x_i) \neq \emptyset$ and $\frac{|NN_{min}(x_i)|}{|NN(x_i)|} < M$ **do**
07. Randomly choose a sample $e_i^l \in NN(x_i)$;
08. Compute the difference $diff = e_i^l - x_i$;
09. Let α = random number between 0 and 1;
10. Compute $x_i^l = x_i + \alpha * diff$;
11. Add x_i^l to S_{newmin};
12. **end if**
13. **end for**
14. **end for**
15. $S_{bmaj} = \bigcup_{x_i \in S_{min}} NN_{maj}(x_i)$;
16. **for** each $y_i \in S_{bmaj}$ **do**
17. Compute k_{maj} nearest neighbors $NN(y_i)$ using Euclidean distance;
18. Divide $NN(y_i)$ to $NN_{maj}(y_i)$ and $NN_{min}(y_i)$;
19. **if** $\frac{|NN_{maj}(y_i)|}{|NN(y_i)|} < N$ **do**
20. Remove y_i from S_{newmaj};
21. **end if**
22. **end for**
23. **return** S_{newmaj} and S_{newmin};

C. LTR Algorithm Selector

LTR is a typical application of machine learning and has a wide range of applications in the field of IR, especially in commercial Web search engines. Liu categorized the existing LTR methods into three types, namely the Pointwise, Pairwise, and Listwise approaches, by their input representation and loss function [21]. In the Pointwise methods, each query-document pair in training data is assumed to have a numerical or ordinal score. Interestingly, some previous studies on SDP have also used the Pointwise approach, approximated by a regression technique, to predict the top-ranked defective software entities [15]-[17]. In addition to the Pointwise approach, we will then introduce the Pairwise and Listwise approaches.

First, the Pairwise approach usually transforms the ranking problem into a binary classification problem. For every two relevant documents for one query, they group into a pair of training entities, each of which is labeled as "+1" or "-1" to tell which one is better in the given pair of documents. Thus, this type of approaches neglects the actual distance between documents in relevance, especially when we focus on the top-*k* relevant documents. There are also a number of well-known implementations of the Pairwise approach, such as RankSVM [43] and RankNet [25].

Second, unlike the Pointwise and Pairwise approaches, the Listwise approach tries to directly optimize a permutation of items in new lists instead of using traditional classification or regression techniques [26]. There are two main optimization ways: one is to directly maximize an evaluation metric like NDCG (Normalized Discounted Cumulative Gain) [44] or MAP (Mean Average Precision), and the other is to minimize the loss function for the whole ordered list. However, the Listwise approach is more complicated than the Pointwise and Pairwise approaches. Even so, we can solve this problem by using various optimization techniques like gradient descent, and some published LTR algorithms have realized the List-wise approach, such as ListNet [26].

Since the performance of each LTR algorithm varies from dataset to dataset, as well as from (evaluation) metric to metric, the primary goal of the LTR algorithm selector is to test and then select the best algorithm for a given dataset regarding commonly-used evaluation metrics.

IV. Experimental Design and Setups

A. Overall Process

To validate the feasibility of our approach in the scenario of CPDP, we elaborately designed an experiment depicted in Fig. 6, including the following four steps. More specifically, we attempted to investigate two research questions:

RQ1. *Does the SMOTE-PENN outperform other competitive data resampling algorithms?*

RQ2. *Which LTR algorithm performs the best for our approach?*

First, we collected raw data from the PROMISE dataset because it has been used in many previous studies on SDP and proved to be of high repeatability for Java projects. A brief introduction to the raw data refers to IV.B.

Second, we conducted the data preprocessing for the raw data using the relevance calculator. This step is a crucial part of our experiment, and more details of relevance labeling refer to IV.C.

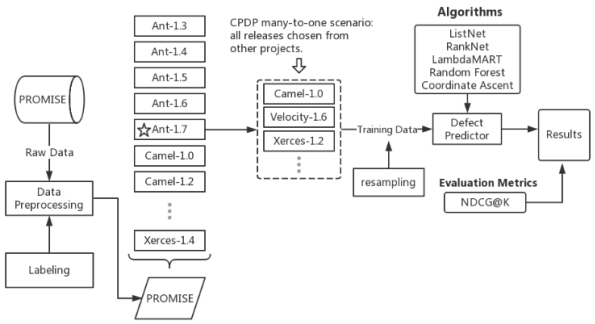

Fig. 6. The overall process of our experiment.

Third, we trained various software defect predictors with the LTR algorithm selector in the *many-to-one* CPDP scenario. For each release of a target project (i.e., the test set, see Ant-1.7), we collected all available releases from other different projects as the training set and used data resampler to tackle the imbalanced data problem. The LTR algorithms used in our experiment refer to IV.D.

Fourth, we made a comparison among those trained defect predictor regarding NDCG to find which LTR algorithm is the best for our approach with different top *k* values. Besides, we employed a nonparametric hypothesis test to examine whether the difference between two defect predictors in performance is statistically significant. More details refer to IV.E.

B. Data Collection

We collected the raw data used in our experiment from the commonly-used PROMISE dataset, including 34 releases from 10 open-source software projects written in Java. For each release, it contains defective and non-defective class files. Each file in a release consists of 20 metrics and a label (i.e., the number of defects). Table I presents the profile of the 34 releases, mainly including the number of files and the ratio of buggy files.

C. Data Preprocessing

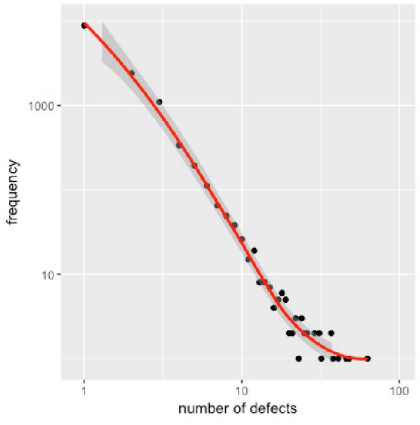

Fig. 7. Frequency distribution of defects and the corresponding fitting curve of the folded normal distribution at a 99% confidence interval.

By using the relevance calculator, we first converted the collected raw PROMISE data into the dataset used in our

experiment. Fig. 7 shows the frequency distribution of defects and the corresponding fitting curve (see the red line) on a log-log plot, where the X-axis represents the number of defects, and the Y-axis denotes the frequency of occurrence. Note that the grey shadow around the red line means a 99% confidence interval. We obtained the approximate estimates of the mean and variance of the dataset, i.e., $\hat{\mu} = 0.0063$ and $\hat{\sigma} = 2.1899$. Interestingly, the form of the fitting curve appears to roughly obey the half-normal distribution, a special kind of the folded normal distribution with mean zero. Based on the mapping rules defined in (8), we then converted each sample's number of defects into its relevance value. Note that this new dataset used in this study is available for download at the website[#].

TABLE I. RAW DATA FROM THE PROMISE DATASET

No.	Release	Project Type	#Files	%Buggyfiles
1	Ant-1.3	Building Java applications	125	16.0%
2	Ant-1.4		178	22.5%
3	Ant-1.5		293	10.9%
4	Ant-1.6		351	26.2%
5	Ant-1.7		745	22.3%
6	Camel-1.0	Open-source integration framework	339	3.8%
7	Camel-1.2		608	35.5%
8	Camel-1.4		872	16.6%
9	Camel-1.6		965	19.5%
10	Ivy-1.1	Dependency manager	111	56.8%
11	Ivy-1.4		241	6.6%
12	Ivy-2.0		352	11.4%
13	Jedit-3.2	Text editor	272	33.1%
14	Jedit-4.0		306	24.5%
15	Lucene-2.0	Search engine library	195	46.7%
16	Lucene-2.2		247	58.3%
17	Lucene-2.4		340	59.7%
18	Poi-1.5	Manipulating file formats	237	59.5%
19	Poi-2.0		314	11.8%
20	Poi-2.5		385	64.4%
21	Poi-3.0		442	63.6%
22	Synapse-1.0	Enterprise service bus	157	10.2%
23	Synapse-1.1		222	27.0%
24	Synapse-1.2		256	33.6%
25	Velocity-1.4	Template engine	196	75.0%
26	Velocity-1.5		214	66.4%
27	Velocity-1.6		229	34.1%
28	Xalan-2.4	Transforming documents	723	15.2%
29	Xalan-2.5		803	48.2%
30	Xalan-2.6		885	46.4%
31	Xerces-init	XML processor	162	47.5%
32	Xerces-1.2		440	16.1%
33	Xerces-1.3		453	15.2%
34	Xerces-1.4		588	74.3%

D. Defect Predictor Training

After obtaining the new dataset composed of the samples with relevance for our experiment, we still have to face the imbalanced data problem. Thus, we resampled the training set for each test set (i.e., a given release) using Algorithm 1 to keep the proportion of normal software entities to defective ones within a reasonable range because defective samples are always less than defect-free samples.

Considering the advantages of the Listwise and Pairwise approaches, we selected two typical Listwise LTR algorithms, ListNet [26] and LambdaMART [45], and one standard Pairwise LTR algorithm, RankNet [25]. Besides, some previous studies [46]-[48] implemented a few new LTR models using random forests and coordinate ascent [49] and showed their generalization for all the three categories of LTR algorithms. According to the obtained training data, we trained different defect predictors using the default configuration of RankLib[*], an open-source library of popular LTR algorithms, to make readers easy to reproduce the results of our experiment. Then, we introduce the five LTR methods in brief.

ListNet (LN): Cao *et al.* [26] first proposed a top-*k* ranking method, ListNet, which represents the probabilities of items being ranked in the top position. Suppose that there are two lists with scores, a Listwise loss function can be defined as the distance between them. ListNet utilizes a neural network as the computing model and then optimizes the Listwise loss function using gradient descent. Unlike RankNet, it always learns directly from the training data with ranking lists of queries instead of document pairs of queries.

LambdaMART (LM): LambdaMART [45] is an improved boosted tree version of LambdaRank, which is built based on RankNet. This algorithm combines LambdaRank and MART (Multiple Additive Regression Tree), a class of boosting algorithms using regression trees to perform gradient descent in the function space [47]. Besides, LambdaMART redefines the gradient of the ranking model and takes into consideration ranking evaluators, which can solve the performance problem of RankNet directly.

RankNet (RN): RankNet was proposed in 2005 by Burges *et al.* [25], and it has been proved to be useful for solving real-world ranking problems. For a given query, RankNet tries to compare any pair of two documents to determine which one is better than the other. After computing ranking probabilities of all document pairs, RankNet minimizes the loss function, the number of Pairwise errors, using gradient descent. Then, the ranking function defined on a three-layer neural network model predicts a score for a rank list.

Random Forests (RF): Random forests are a useful ensemble learning approach to classification, regression, and other tasks. Since each tree can maximize the objective in a coordinate-wise fashion, this algorithm works well in both the Pointwise and Listwise scenarios. For example, Ibrahim *et al.* [47] proposed an RF-based Listwise algorithm, which directly optimizes an IR metric of choice using the greedy approach. Besides, Jiang [46] reported the feasibility of the RF-based Pairwise approach. These previous studies suggest that RF could be an underlying learning algorithm for training ranking predictors.

Coordinate Ascent (CA): Coordinate ascent is a standard technique for multi-variate objective optimization problems [49]. It is often deemed to be a Listwise LTR algorithm that is capable of optimizing any IR measure directly. Coordinate ascent repeatedly cycles through each of the parameters and minimizes over it while fixing all other parameters until there is no detectable change to the free parameter. Although this

[#] https://github.com/ssea-lab/PROMISE

[*] https://sourceforge.net/p/lemur/wiki/RankLib/

algorithm has been implemented in RankLib, it is known to converge slowly on objective functions with long ridges [49].

E. Evaluation Method

In the IR field, three commonly-used evaluation metrics, *Precision*, *Recall*, and F_1 *score*, measure the efficiency of ranking sets of disordered documents, and MAP and NDCG are suitable for ordered permutations. Considering the goal of this work, we evaluate the ranking results of different LTR algorithms in terms of NDCG that also considers the relative position information. Another reason why we chose NDCG is that some Listwise LTR algorithms such as ListNet and LambdaMART optimize NDCG directly.

$$N(k) = Z_k \cdot \sum_{i=1}^{k} \frac{2^{rel_i}-1}{\log_2(i+1)}, \quad (9)$$

where rel_i is the graded relevance of the result at position i and Z_k is a normalization factor. Here, $rel_i \in \{r_i\}$ (see (8)).

For each target release selected as the test set, we can get an evaluation result regarding NDCG. Take Ant-1.7 (see Fig. 6) as an example. For this release, we collect only the other 29 releases that do not belong to the Ant project (see Table I) as the training set. After processing the training data using a data resampling algorithm (e.g., SMOTE-PENN), we can train a new defect predictor with an LTR algorithm. Then, we obtain an evaluation result of the predictor concerning Ant-1.7 in terms of NDCG. To make a comprehensive comparison of different LTR algorithms, we collected and analyzed all 34 evaluation results for each LTR algorithm under discussion. To test whether two groups of evaluation results originate from the same distribution, we also utilized a non-parametric hypothesis test, namely the Wilcoxon signed-rank test [50], which was performed with a significance level $\alpha = 0.05$ according to the null hypothesis that there is no statistically significant difference between the two groups.

V. Experimental Results

A. Answer to RQ1

As with the previous studies [21], [22], [43], [47], [48], we set the maximum value of *k* to ten. In addition to SMOTE-PENN, we selected the other six competitive data resampling algorithms, i.e., CNN (Condensed Nearest Neighbor), RENN (Repeated Edited Nearest Neighbor), ROS (Random Over-sampling), RUS (Random Under-sampling), SMOTE [41], and TL (Tomek Link). Note that all the six algorithms were implemented by using the imbalanced-learn[+] with default parameter settings.

TABLE II. Comparison of Seven Resampling Algorithms

LTR	Mean value of NDCG@10						
	RUS	***CNN***	***RENN***	***TL***	***ROS***	***SMOTE***	***Ours***
LM	0.479	0.435	0.431	0.462	0.436	0.495	**0.497**
RF	0.580	0.479	**0.590**	0.547	0.534	0.509	0.573
CA	0.539	0.360	0.526	0.516	0.371	0.483	**0.552**
RN	0.704	0.669	0.721	0.705	0.690	0.691	**0.729**
LN	0.653	0.662	0.683	0.679	0.678	0.682	**0.717**

Each cell in Table II represents the average of all 34 evaluation results, which were obtained by the corresponding LTR algorithm (in the row) and data resampling algorithm (in the column). As shown in Table II, SMOTE-PENN takes the first place four times (see the bold numbers in the last column of this table). Also, it gets a third place on RF. Hence, on the whole, SMOTE-PENN performs better than the other six data resampling algorithms in the framework of our approach, according to the mean value of NDCG@10.

The SMOTE-PENN algorithm outperforms the other six competitive data resampling algorithms in the framework of our approach.

B. Answer to RQ2

As mentioned above, we trained five types of software defect predictor using the proposed approach and calculated an NDCG score of a given predictor for each target release. To analyze the prediction performance of various types of defect predictors better, we set three values for *k*, namely three, five, and ten, in our experiment. Tables III, IV, and V describe the overall prediction performance of the five types of defect predictors regarding NDCG@3, NDCG@5, and NDCG@10, respectively. We present the distributions of the metric values using notched box plots. Note that notched box plots are one of the most common box plots in descriptive statistics. Unlike traditional box plots, notched box plots use a "notch" of the box around the median to provide a roughly visible guide to the significance of the difference of medians. The width of the notches is proportional to the interquartile range of a given sample and inversely proportional to the square root of the size of the sample.

TABLE III. Performance Comparison Regarding NDCG@3

LTR	NDCG@3		Wilcoxon signed-rank test			
	Median	*Notched Boxplot*	*LM*	*RF*	*CA*	*RN*
LM	0.518					
RF	0.701		○			
CA	0.701		○	—		
RN	**0.759**		√	○	○	
LN	0.750		√	○	○	—

√: very significant difference ($p < 0.01$), ○: significant difference ($p < 0.05$), and —: no statistically significant difference (i.e., accepting the null hypothesis)

TABLE IV. Performance Comparison Regarding NDCG@5

LTR	NDCG@5		Wilcoxon signed-rank test			
	Median	*Notched Boxplot*	*LM*	*RF*	*CA*	*RN*
LM	0.481					
RF	0.639		○			
CA	0.631		○	—		
RN	**0.745**		√	○	√	
LN	0.724		√	○	○	—

√: very significant difference ($p < 0.01$), ○: significant difference ($p < 0.05$), and —: no statistically significant difference (i.e., accepting the null hypothesis)

TABLE V. Performance Comparison Regarding NDCG@10

LTR	NDCG@10		Wilcoxon signed-rank test			
	Median	*Notched Boxplot*	*LM*	*RF*	*CA*	*RN*
LM	0.470					
RF	0.599		○			
CA	0.565		○	—		
RN	**0.740**		√	√	√	
LN	0.736		√	√	√	—

√: very significant difference ($p < 0.01$), ○: significant difference ($p < 0.05$), and —: no statistically significant difference (i.e., accepting the null hypothesis)

[+] http://contrib.scikit-learn.org/imbalanced-learn/stable/

As shown in Tables III, IV, and V, RN performs the best for our approach, followed by LN, according to the median value of 34 evaluation results. It is worth mentioning that the prediction performance of the five types of defect predictors tend to decrease with an increase in the value of *k*. When *k* is equal to three, the median values of these predictors except LM are higher than 0.7, suggesting that they are capable of identifying the top three severe software entities. Even so, there are also significant differences between any one of LM, RF, and CA and either RN or LN. Along with the increase in the value of *k*, the differences mentioned above are more noticeable. However, there is no statistically significant difference between LN and RN regardless of the value of *k*, although RN has a higher median value than LN.

To better understand the ranking process of our approach, we also present a real case study of LN on the release Xerces-1.4 in Table VI. For this release, the NDCG@10 value of the defect predictor built with LN and SMOTE-PENN reaches 0.815, indicating a pretty good prediction result. In fact, the predictor has a higher value of NDCG@3. However, the two classes, *CMNode* and *DocumentFragment*, are overvalued in this ordered list, thus resulting in apparent decreases in NDCG@5 and NDCG@10 compared with their ideal values. On the other hand, the performance degradation of the defect predictor is also because two severe classes with high graded relevance, *ParentNode* and *XSDGroupTraverser*, are ranked lower than those less severe classes.

TABLE VI. CASE STUDY OF PREDICTING THE TOP TEN DEFECTIVE FILES

Rank (i)	Class Name	Bug number	Relevance score (rel_i)	$\log_2(i+1)$	$\frac{rel_i}{\log_2(i+1)}$	DCG_i
1	NodeImpl	16	3	1	3	3
2	XSAttributeChecker	13	3	1.585	1.893	4.893
3	BaseMarkupSerializer	15	3	2	1.5	6.393
4	CMNode	2	1	2.32	0.431	6.823
5	DOMASWriter	4	2	2.585	0.774	7.597
6	XSDElementTraverser	10	3	2.807	1.069	8.666
7	TreeWalkerImpl	3	2	3	0.667	9.332
8	DocumentFragment	0	0	3.170	0	9.332
9	XSDGroupTraverser	8	3	3.322	0.903	10.236
10	ParentNode	12	3	3.459	0.867	11.102

The RankNet algorithm performs the best for our approach. However, there is no statistically significant difference between it and the ListNet algorithm.

VI. DISCUSSION

A. Impact of Top-k

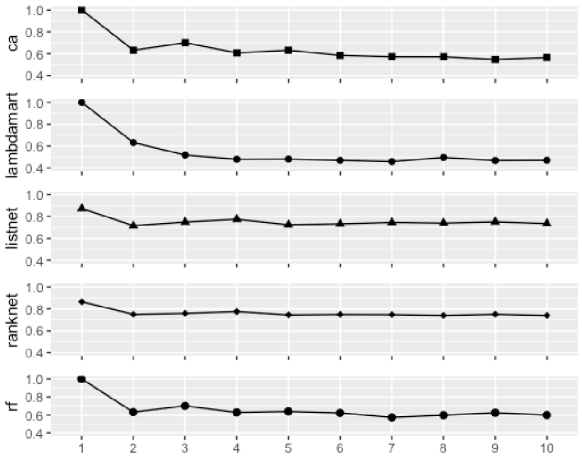

Fig. 8. Changing trend of NDCG values ($k \in [1,10]$).

Because we revisit SDP as a top-*k* ranking problem in this work, the parameter *k* may affect the prediction performance of our approach. As with many previous studies in the IR field, Fig. 8 depicts the changing trends of the five types of defect predictors regarding NDCG when *k* increases from one to ten. Generally speaking, RN and LN show relatively stable and good prediction performance irrespective of the value of *k*. Note that although all the five types of defect predictors have the highest value of NDCG@1, this parameter (top-1) has no practical significance to our study. Thus, we do not use it to evaluate the performance of the defect predictors discussed in Section V.

B. Potential Application in SDP

TABLE VII. PERFORMANCE COMPARISON IN DIFFERENT CONTEXTS REGARDING MEDIAN NDCG@3

LTR	WPDP-Single	WPDP-All	Ours
CA	0.549	0.639	**0.701**
LM	**0.563**	0.469	0.518
LN	0.700	0.708	**0.750**
RN	0.717	0.717	**0.759**
RF	0.613	0.648	**0.701**

TABLE VIII. PERFORMANCE COMPARISON IN DIFFERENT CONTEXTS REGARDING MEDIAN NDCG@5

LTR	WPDP-Single	WPDP-All	Ours
CA	0.524	0.515	**0.631**
LM	0.480	**0.525**	0.481
LN	0.691	0.689	**0.724**
RN	0.704	0.705	**0.745**
RF	0.574	**0.675**	0.639

TABLE IX. PERFORMANCE COMPARISON IN DIFFERENT CONTEXTS REGARDING MEDIAN NDCG@10

LTR	WPDP-Single	WPDP-All	Ours
CA	0.471	0.549	**0.565**
LM	0.515	**0.534**	0.470
LN	0.701	0.702	**0.736**
RN	0.699	0.718	**0.740**
RF	0.576	**0.629**	0.599

Previous studies [1]-[4], [9] suggest that the performance of CPDP does not compare to that of WPDP, especially for binary classification-based approaches. Thus, a few scholars criticized them for lacking practical value in software quality assurance. To test the possible applicability of the proposed approach to real software projects, we further performed a comparison for it in different CPDP and WPDP scenarios. We present the evaluation results regarding NDCG@3, 5, and 10 in Tables VII, VIII, and IX, respectively. In the three tables, *WPDP-Single* represents that we use the most recent release (from the same project) to the test set as the training set, while *WPDP-All* indicates that the training set contains all of the available historical releases from the same project except the test set. Take Ant-1.7 as an example. We choose Ant-1.6 and the whole set of Ant-1.3, 1.4, 1.5, and 1.6 to train a new defect predictor in the contexts of WPDP-Single and WPDP-All, respectively. Note that we did not apply any data resampling technique to the contexts of WPDP-Single and WPDP-All since the size of the training data is not big enough.

In general, the five types of defect predictors performed better in the context of WPDP-All than they did in the context of WPDP-Single. In the *many-to-one* CPDP scenario, more heterogeneous releases from other different software projects were used to train defect predictors. As a result, the predictors trained by our approaches in such a scenario won ten out of fifteen times, indicating that their prediction performance is indeed comparable to or even better than those obtained in the WPDP scenario. Thus, our work could lay a foundation for efficient search engines for severe entities in real-world software testing activities without local historical data and may spark new research interest in the traditional field of SDP.

C. Threats to Validity

Although we have presented exciting results of this work, there exist some possible threats to the validity of our results, mainly including the internal validity and external validity.

The threats to the *internal validity* of our study include three main aspects: relevance conversion, data resampling, and parameter setting. First, we calculated the relevance of a given software entity to the query based on the folded normal distribution. However, the frequency distribution of defects (see Fig. 7) follows a power law with a higher fitting degree. We argue that the folded normal distribution can better model graded relevance because power-law distributions are scale-free. Second, considering the imbalanced data problem, we processed the training data using data resampling techniques. Although data resampling has been proved to be useful for SDP, we have to admit that these resampling techniques may introduce randomness into our experiment. Third, to be fair, we trained all the defect predictors using the default settings of RankLib without specified optimization. That is to say, the fine-tuning of these LTR algorithms may change our results.

The *external validity* concerns the generalizability of a study to other situations. First, the performance of our method on other datasets such as AEEEM is yet to be tested. Besides, for a new dataset, we must first convert it into a set of query-document pairs using the relevance calculator of our method. Second, since we selected only five common LTR algorithms, the merit of RankNet over other LTR algorithms for cross-project SDP remains unknown. Third, this work reports the high applicability of our approach in the *many-to-one* CPDP scenario based on the assumption that we can collect releases from other different projects as many as possible. Therefore, our method performs better in this scenario than it does in the case of a small amount of local historical data. In other words, this conclusion does not apply to the situation that there is sufficient within-project data.

VII. CONCLUSION AND FUTURE WORK

In the SDP field, the CPDP problem is generally treated as a binary classification problem or a regression problem. However, it is difficult for these existing CPDP approaches to detect a ranking of the most severe entities in practical software quality assurance activities. To meet the real needs of software developers, in this paper, we present a top-*k* LTR approach to CPDP, including a relevance calculator, a data resampler, and an LTR algorithm selector. For the relevance calculator, we first convert the number of defects into graded relevance to a specific query according to the three-sigma rule. Then, we propose a new data resampling method called SMOTE-PENN. Experimental results indicate that SMOTE-PENN outperforms the other six competitive data resampling algorithms regarding NDCG@10. Also, RankNet performs the best for our approach regarding NDCG@3, 5, and 10, but there is no statistically significant difference between it and ListNet according to the Wilcoxon signed-rank test.

Our work is the first attempt to revisit the CPDP problem from the perspective of top-*k* ranking. To further improve the performance and applicability of the proposed approach, our future studies are twofold. First, we will generalize our work to other situations with more datasets and more new LTR algorithms. Second, we will extend the approach framework to support efficient software entity search based on multiple queries (rather than a single query in this paper) and better evaluation metric optimization algorithms.

ACKNOWLEDGMENT

This work was supported by the National Basic Research Program of China (No. 2014CB340404), National Key Research and Development Program of China (No. 2017YFB1400602), National Science Foundation of China (Nos. 61672387 and 61702378), and Wuhan Yellow Crane Talents Program for Modern Services Industry. Yutao Ma is the corresponding author of this paper.

REFERENCES

[1] C. Catal, "Software fault prediction: A literature review and current trends," Expert Syst. Appl., vol. 38, no. 4, pp. 4626–4636, 2011.

[2] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," IEEE T. Software Eng., vol. 38, no. 6, pp. 1276–1304, 2012.

[3] G. Abaei and A. Selamat, "A survey on software fault detection based on different prediction approaches," Vietnam J. Comput. Sci., vol. 1, no. 2, pp. 79–95, 2014.

[4] R. Malhotra, “A systematic review of machine learning techniques for software fault prediction,” Appl. Soft Comput., vol. 27, pp. 504–518, 2015.

[5] R. S. Wahono, “A Systematic Literature Review of Software Defect Prediction: Research Trends, Datasets, Methods and Frameworks,” J. Softw. Eng., vol. 1, no. 1, pp. 1–16, 2015.

[6] S. Herbold, A, Trautsch, and J. Grabowski, “A Comparative Study to Benchmark Cross-project Defect Prediction Approaches,” IEEE Trans. Software Eng., pp. 1–25, 2017. doi: 10.1109/TSE.2017.2724538.

[7] S. S. Rathore and S. Kumar, “A study on software fault prediction techniques,” Artif. Intell. Rev., pp. 1–73, 2017, doi: 10.1007/s10462-017-9563-5.

[8] Z. Li, X.-Y. Jing, and X. Zhu, “Progress on approaches to software defect prediction,” IET Softw., vol. 12, no. 3, pp. 161–175, 2018.

[9] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, “An empirical study on software defect prediction with a simplified metric set,” Inform. Software Tech., vol. 59, pp. 170–190, 2015.

[10] S. J. Pan and Q, Yang, “A Survey on Transfer Learning,” IEEE T. Knowl. Data En., vol. 22, no. 10, pp. 1345–1359, 2010.

[11] S. Herbold, “A systematic mapping study on cross-project defect prediction,” arXiv, arXiv:1705.06429, 2017.

[12] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, B. Xu, “How Far We Have Progressed in the Journey? An Examination of Cross-Project Defect Prediction,” ACM T. Softw. Eng. Meth., vol. 27, no. 1, p. 1, 2018.

[13] F. Porto, L. Minku, E. Mendes, and A. Simao, “A Systematic Study of Cross-Project Defect Prediction With Meta-Learning,” arXiv, arXiv:1802.06025, 2018.

[14] M. Chen and Y. Ma, “An empirical study on predicting defect numbers,” in Proceedings of SEKE’15. Pittsburgh: KSI Research Inc., 2015, pp. 397–402.

[15] G. You and Y. Ma, “A Ranking-Oriented Approach to Cross-Project Software Defect Prediction: An Empirical Study,” in Proceedings of SEKE’16. Pittsburgh: KSI Research Inc., 2016, pp. 159–164.

[16] G. You, F. Wang, and Y. Ma, “An Empirical Study of Ranking-Oriented Cross-Project Software Defect Prediction,” Int. J. Softw. Eng. Know., vol. 26, no. 9&10, pp. 1511–1538, 2016.

[17] X. Yang, K. Tang, and X. Yao, “A Learning-to-Rank Approach to Software Defect Prediction,” IEEE T. Reliab., vol. 64, no. 1, pp. 234–246, 2015.

[18] S. S. Rathore and S. Kumar, “An empirical study of some software fault prediction techniques for the number of faults prediction,” Soft Comput., vol. 21, no. 24, pp. 7417–7434, 2017.

[19] S. S. Rathore and S. Kumar, “Towards an ensemble based system for predicting the number of software faults,” Expert Syst. Appl., vol. 82, pp. 357–382, 2017.

[20] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, and S. Ye, “Learning from Imbalanced Data for Predicting the Number of Software Defects,” in Proceedings of IEEE ISSRE’17. New York: IEEE Computer Society, 2017, pp. 78–89.

[21] T.-Y. Liu, “Learning to Rank for Information Retrieval,” Found. Trends Inf. Ret., vol. 3, no. 3, pp. 225–331, 2009.

[22] F. Xia, T.-Y. Liu, and H. Li, “Statistical Consistency of Top-k Ranking,” in Proceedings of NIPS’09. La Jolla: Neural Information Processing Systems Foundation, Inc., 2009, pp. 2098–2106.

[23] N. Tax, S. Bockting, and D. Hiemstra, “A cross-benchmark comparison of 87 learning to rank methods,” Inform. Process. Manag., vol. 51, no. 6, pp. 757–772, 2015.

[24] M. Jureczko and L. Madeyski, “Towards identifying software project clusters with regard to defect prediction,” in Proceedings of PROMISE’10. New York: ACM Press, 2010, p. 9.

[25] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. N. Hullender, “Learning to rank using gradient descent,” in Proceedings of ICML’05. New York: ACM Press, 2005, pp. 89–96.

[26] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, “Learning to rank: from pairwise approach to listwise approach,” in Proceedings of ICML’07. New York: ACM Press, 2007, pp. 129–136.

[27] J. E. Gaffney Jr., “Estimating the Number of Faults in Code,” IEEE T. Software Eng., vol. 10, no. 4, pp. 459–465, 1984.

[28] N. E. Fenton and M. Neil, “A Critique of Software Defect Prediction Models,” IEEE T. Software Eng., vol. 25, no. 5, pp. 675–689, 1999.

[29] X. Yu, Z. Ma, C. Ma, Y. Gu, R. Liu, and Y. Zhang, “FSCR: A Feature Selection Method for Software Defect Prediction,” in Proceedings of SEKE’17. Pittsburgh: KSI Research Inc., 2017, pp. 351–356.

[30] T. T. Nguyen, T. Q. An, V. T. Hai, and T. M. Phuong, “Similarity-based and rank-based defect prediction,” in Proceedings of ATC’14. New York: IEEE Press, 2014, pp. 321–325.

[31] M. D'Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in Proceedings of MSR’10. New York: IEEE Computer Society, 2010, pp. 31–41.

[32] F. C. Leone, R. B. Nottingham, and L. S. Nelson, “The Folded Normal Distribution,” Technometrics, vol. 3, no. 4, pp. 543–550, 1961.

[33] N. L. Johnson, “The folded normal distribution: accuracy of the estimation by maximum likelihood,” Technometrics, vol. 4, no. 2, pp. 249–256, 1962.

[34] M. Tsagris, C. Beneki, and H. Hassani, “On the folded normal distribution,” Mathematics, vol. 2, no. 1, pp. 12–28, 2014.

[35] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum Likelihood from Incomplete Data via the EM Algorithm,” J. R. Stat. Soc. B, vol. 39, no. 1, pp. 1–38, 1977.

[36] D. C. Montgomery, Introduction to Statistical Quality Control (5 ed.). Hoboken, New Jersey: John Wiley & Sons, 2005.

[37] T. M. Khoshgoftaar, K. Gao, and N. Seliya, “Attribute Selection and Imbalanced Data: Problems in Software Defect Prediction,” in Proceedings of IEEE ICTAI’10. New York: IEEE Computer Society, 2010, vol. 1, pp. 137–144.

[38] S. Wang and X. Yao, “Using Class Imbalance Learning for Software Defect Prediction,” IEEE T. Reliab., vol. 62, no. 2, pp. 434–443, 2013.

[39] Q. Song, Y, Guo, and M. Shepperd, “A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction,” IEEE T. Software Eng., pp. 1–16, 2018. doi: 10.1109/TSE.2018.2836442.

[40] I. Tomek, “An experiment with the edited nearest-neighbor rule,” IEEE T. Syst. Man Cy., vol. SMC-6, no. 6, pp. 448–452, 1976.

[41] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: Synthetic Minority Over-sampling Technique,” J. Artif. Intell. Res., vol. 16, pp. 321–357, 2002.

[42] H. Han, W. Wang, and B. Mao, “Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning,” in Proceedings of ICIC’05. Heidelberg: Springer, 2005, vol. 1, pp. 878–887.

[43] T. Joachims, “Optimizing search engines using clickthrough data,” in Proceedings of ACM KDD’02. New York: ACM Press, 2002, pp. 133–142.

[44] K. Jarvelin and J. Kekalainen, “Cumulated gain-based evaluation of IR techniques,” ACM T. Inform. Syst., vol. 20, no. 4, pp. 422–446, 2002.

[45] C. J.C. Burges, “From RankNet to LambdaRank to LambdaMART: An Overview,” Microsoft Research Technical Report, MSR-TR-2010-82, 2010.

[46] L. Jiang, “Learning random forests for ranking,” Front. Comput. Sci. Chi., vol. 5, no. 1, pp. 79–86, 2011.

[47] M. Ibrahim and M. J. Carman, “Comparing Pointwise and Listwise Objective Functions for Random-Forest-Based Learning-to-Rank,” ACM T. Inform. Syst., vol. 34, no. 4, pp. 20:1–20:38, 2016.

[48] I. Uysal and W. B. Croft, “User oriented tweet ranking: a filtering approach to microblogs,” in Proceedings of ACM CIKM’11. New York: ACM Press, 2011, pp. 2261–2264.

[49] D. Metzler and W. B. Croft, “Linear feature-based models for information retrieval,” Inform. Retrieval, vol. 10, no. 3, pp. 257–274, 2007.

[50] F. Wilcoxon, “Individual Comparisons by Ranking Methods,” Biometrics Bull., vol. 1, no. 6, pp. 80–83, 1945.