9-2018

Rule-based specification mining leveraging learning to rank

Zherui CAO
Zhejiang University

Yuan TIAN
Singapore Management University, ytian@smu.edu.sg

Bui Tien Duy LE
Singapore Management University, btdle@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Citation

CAO, Zherui; TIAN, Yuan; LE, Bui Tien Duy; and LO, David. Rule-based specification mining leveraging learning to rank. (2018). *Automated Software Engineering*. 25, (3), 501-530.
Available at: https://ink.library.smu.edu.sg/sis_research/3988

Published in Automated Software Engineering, 2018 September, 25 (3), 501-530.
https://doi.org/10.1007/s10515-018-0231-z

Rule-based specification mining leveraging learning to rank

Zherui Cao[1] · Yuan Tian[2] · Tien-Duy B. Le[2] · David Lo[2]

Abstract Software systems are often released without formal specifications. To deal with the problem of lack of and outdated specifications, rule-based specification mining approaches have been proposed. These approaches analyze execution traces of a system to infer the rules that characterize the protocols, typically of a library, that its clients must obey. Rule-based specification mining approaches work by exploring the search space of all possible rules and use interestingness measures to differentiate specifications from false positives. Previous rule-based specification mining approaches often rely on one or two interestingness measures, while the potential benefit of combining multiple available interestingness measures is not yet investigated. In this work, we propose a learning to rank based approach that automatically learns a good combination of 38 interestingness measures. Our experiments show that the learning to rank based approach outperforms the best performing approach leveraging single interestingness measure by up to 66%.

Zherui Cao and Yuan Tian have contributed equally to this work.

This work was done while the author was visiting Singapore Management University.

✉ Yuan Tian
ytian@smu.edu.sg

Zherui Cao
caozherui@zju.edu.cn

Tien-Duy B. Le
btdle.2012@smu.edu.sg

David Lo
davidlo@smu.edu.sg

1 College of Computer Science and Technology, Zhejiang University, Hangzhou, China

2 School of Information Systems, Singapore Management University, Singapore, Singapore

Keywords Specification mining · Learning to rank · Automated software development · Software maintenance and evolution

1 Introduction

Ideally, a software system should have clearly documented specifications to avoid, detect and correct bugs. However, due to the hard deadlines and agile nature of many projects, software is often released and evolved without clear and complete specifications. Formal specifications are especially lacking since developers often do not have the necessary skill or motivation to write them (Knight et al. 1997). Even if a system includes formal specifications, these specifications can quickly get outdated as a software evolves (Zhong and Su 2013). To address the issue of the lack of and outdated specifications, specification mining approaches have been proposed (Yang et al. 2006; Lo and Khoo 2006; Lo et al. 2008, 2012; Li et al. 2010; Beschastnikh et al. 2011; Lo and Maoz 2012; Krka et al. 2014; Le et al. 2015; Le and Lo 2015; Lemieux et al. 2015).

This paper focuses on a family of specification mining techniques, namely *rule-based specification mining*, that analyze execution traces of systems and infer specifications in the form of rules, e.g., "whenever IoAcquireRemoveLock is called, IoReleaseRemoveLock must eventually be called", "whenever File.read() is called, File.open() must have been called before", etc. Dwyer et al. refer to these rules as response and precedence patterns, and they correspond to widely used temporal logic expressions for verification purpose (Dwyer et al. 1999). Since these rules describe temporal ordering of events (i.e., method calls), they are often referred to as *temporal rules* (Lo et al. 2008, 2012). These specification rules are valuable and have been used for a wide range of practical tasks, such as debugging (Gabel and Su 2010; Microsoft 2016), malware detection (Yang et al. 2014), and data structure repair (Demsky et al. 2006).

Rule-based specification mining approaches typically work by enumerating candidate rules in a search space of all possible rules, and evaluating the likelihood of each rule to be true based on some interestingness measures (Safyallah and Sartipi 2006; Yang et al. 2006; Lo et al. 2008; Lemieux et al. 2015). Among these interestingness measures, *Support* and *Confidence* are the most frequently considered ones. The *Support* measure calculates the number of times a candidate rule is satisfied in the execution traces, and the *Confidence* measure calculates how likely the post-condition of a rule is followed, when its pre-condition occurred in an execution trace. Recently, Le and Lo conducted an empirical study to examine the performance of applying 38 different interestingness measures to identify specifications for a set of classes in the Java SDK (Le and Lo 2015). They found that *Support* and *Confidence* mediocrely performed among the 38 measures, and they are unable to rank many correct rules before incorrect ones. However, in Le and Lo's empirical study, they only considered identifying rules based on a *single measure*, which leaves an important question unanswered, i.e., could we compose all the 38 available interestingness measures together to improve the performance of automated rule-based specification mining based on a single interestingness measure? This question motivates our study.

In this study, we propose a learning to rank based approach to compose the 38 measures together. Learning to rank is a machine learning technique that learns an appropriate combination of features for ranking objects (Liu 2009). Specifically, for the ranking engine of our approach, we consider eight learning to rank algorithms to learn a composition of the 38 measures based on a training set of classes, whose specifications are known. The composite measures are then used to identify correct specifications for a new system or library given its execution traces.

We have evaluated the effectiveness of our learning to rank based approach for identifying correct specifications of a number of popular API classes previously considered by Le and Lo (2015). We use execution traces generated by running the DaCapo benchmark (Blackburn et al. 2006) and input these to our approach and the baselines. We find that the best-performing learning to rank approach can achieve much better performance (i.e., up to 66.22% improvement) than the best performance achieved by using a single measure alone.

The contributions of this paper include:

1. We propose a learning to rank approach to measure the likelihood of a candidate rule being a specification leveraging 38 interestingness measures. This work extends the prior work by Le and Lo (2015), which is the first to use the 38 measures for specification mining.
2. We investigate the effect of varying learning to rank algorithms for our approach. Our experiments demonstrate that the best performing variant of our approach can improve the best performing baseline by up to 66%.

The rest of this paper is organized as follows. In Sect. 2, we briefly introduce rule-based specification mining and list the 38 interestingness measures considered in this work. In Sect. 3, we describe the details of our learning to rank based specification mining approaches. In Sect. 4, we present the settings and results of experiments that evaluate the effectiveness of our proposed approach. We discuss additional topics in Sect. 5. Related works are presented in Sect. 6. We conclude and mention future work in Sect. 7.

2 Rule-based specification mining: a primer

In this section, we briefly introduce background materials on temporal rules (Sect. 2.1) and interestingness measures (Sect. 2.2).

2.1 Temporal rules

A temporal rule is defined to contain a series of events that are observed in program execution traces. Each rule represents a particular property between two or more events and possibly be expressed in *linear temporal logic* (LTL). We list the definitions of some important notations of LTL in Table 1.

In this study, similar to Le and Lo (2015) and Yang et al. (2006), we focus on inferring two-event temporal rules since these rules the most commonly used specification

Table 1 Some linear temporal logics notation. Φ and Ω correspond to a linear temporal logic expression

Operator name	Usage	Description
G (Globally)	G Φ	Φ must hold at every point along the execution trace
X (Next)	X Φ	Φ must hold at the next point of the current point
F (Finally)	F Φ	Φ eventually must hold at a point along the execution trace
W (Week Until/Unless)	Φ W Ω	Φ holds until Ω hold. As long as Φ does not hold, Ω must hold

rules. We focus on two temporal rules relating two events (i.e., method calls) A and B:

1. A is **always followed by** B (denoted by A→B): an occurrence of event A must be eventually followed by an event B in the execution trace. For example, whenever an `android.os.PowerManager.WakeLock` object is acquired by calling `acquire()` method, eventually the `WakeLock` object has to be released by calling `release()` method.[1] We refer to A→B as a *forward-eventually rule* where A is the precondition and B is the postcondition of the rule. In LTL, this rule is expressed as: $\mathbf{G}(\mathrm{A} \rightarrow \mathbf{XF}\ \mathrm{B})$.
2. A is **always preceded by** B (denoted by B←A): an occurrence of event A must be preceded by event B in the execution trace. For example, every invocation of a non-static method `foo` of a Java object must be preceded by the initialization of that object (i.e., `init`←`foo`). We refer to B←A as a *backward-eventually rule* where A is the precondition and B is the postcondition of the rule. In LTL, the rule is expressed as: $\neg\mathrm{A}\ \mathbf{W}\ \mathrm{B}$.

2.2 Interestingness measures

Most rule-based specification mining approaches employ traditional interestingness measures (i.e., support and confidence) to help differentiate correct specifications from spurious rules (Fahland et al. 2013; Li and Zhou 2005; Lo et al. 2008, 2012; Lo and Maoz 2012; Yang et al. 2006). Nevertheless, besides support and confidence, many other interestingness measures have been proposed in statistics, data mining, and machine learning areas (Geng and Hamilton 2006). For instances, odds ratio is often used to indicate the odds of a particular event to occur (e.g., a health outcome) after a particular medical treatment has been given or an exposure to a particular substance (Stampfer 2009; Henning and Pfeiffer 2009). Prevalence is often adopted in epidemiology (Rothman 2012). For example, it has been used to study a proportion of a population that is affected by a disease (Mutegi et al. 2009). These measures are potentially applicable to mine specifications. Recently, Le and Lo have evaluated the performance of 38 interestingness measures including support and confidence for inferring temporal rules (Le and Lo 2015). Their comparative study indicates that odds ratio, and other measures outperform support and confidence in many cases.

[1] https://developer.android.com/training/scheduling/wakelock.html.

Table 2 Interestingness measures—part I

ID	Interestingness measure	Formula	Range
M_1	Support	$P(AB)$	[0, 1]
M_2	Confidence/ Precision	$P(B \mid A)$	[0, 1]
M_3	Coverage	$P(A)$	[0, 1]
M_4	Prevalence	$P(B)$	[0, 1]
M_5	Recall	$P(A \mid B)$	[0, 1]
M_6	Specificity	$P(\neg B \mid \neg A)$	[0, 1]
M_7	Accuracy	$P(AB) + P(\neg A \neg B)$	[0, 1]
M_8	Lift/Interest	$\frac{P(AB)}{P(A)P(B)}$	$[0, +\infty)$
M_9	Leverage	$P(B \mid A) - P(A)P(B)$	$[-1, 1]$
M_{10}	Added value/ Change of support	$P(B \mid A) - P(B)$	$[-1, 1]$
M_{11}	Relative risk	$\frac{P(B \mid A)}{P(B \mid \neg A)}$	$[0, +\infty)$
M_{12}	Jaccard	$\frac{P(AB)}{P(A)+P(B)-P(AB)}$	$(-\infty, +\infty)$
M_{13}	Certainty factor	$\frac{P(B \mid A)-P(B)}{1-P(B)}$	$(-\infty, +\infty)$
M_{14}	Odds ratio	$\frac{P(AB)P(\neg A \neg B)}{P(A \neg B)P(\neg BA)}$	$[0, +\infty)$
M_{15}	Yule's Q	$\frac{P(AB)P(\neg A \neg B)-P(A \neg B)P(\neg AB)}{P(AB)P(\neg A \neg B)+P(A \neg B)P(\neg AB)}$	$(-\infty, +\infty)$
M_{16}	Yule's Y	$\frac{\sqrt{P(AB)P(\neg A \neg B)}-\sqrt{P(A \neg B)P(\neg AB)}}{\sqrt{P(AB)P(\neg A \neg B)}+\sqrt{P(A \neg B)P(\neg AB)}}$	$(-\infty, +\infty)$
M_{17}	Klosgen	$\sqrt{P(AB)} \times \max(P(B \mid A) - P(B), P(A \mid B) - P(A))$	$[-1, 1]$
M_{18}	Conviction	$\frac{P(A)P(\neg B)}{P(A \neg B)}$	$[0, +\infty)$
M_{19}	Interestingness weighting	$((\frac{P(AB)}{P(A)P(B)})^k - 1) \times P(AB)^m$	$[0, +\infty)$
	Dependency	(We assume $k = 2$ and $m = 2$)	
M_{20}	Collective strength	$\frac{P(AB)+P(\neg B \mid \neg A)}{P(A)P(B)+P(\neg A)P(\neg B)} \times \frac{1-P(A)P(B)-P(\neg A)P(\neg B)}{1-P(AB)-P(\neg B \mid \neg A)}$	$(-\infty, +\infty)$
M_{21}	Laplace correction	$\frac{N(AB)+1}{N(A)+2}$	[0.5, 1]

Tables 2 and 3 list the definitions and ranges of the 38 interestingness measures that are considered by Le and Lo (2015). These measures are formulated based on probabilities, and their outputs correspond to the interestingness of a rule consisting of two parts A and B, where A is the precondition and B is the postcondition of the rule (Geng and Hamilton 2006). In the two tables, $P(A)$ is the probability that precondition A occurs; similarly, $P(B)$ is the probability that postcondition B occurs etc. The other symbols follow standard probability notations. For example, *support* is defined as the probability of A and B to occur together (i.e., the proportion of instances where the precondition A is followed by the postcondition B); similarly,

Table 3 Interestingness measures—part II

ID	Interestingness measure	Formula	Range
M_{22}	Gini index	$P(A) \times (P(B \mid A)^2 + P(\neg B \mid A)^2)+$ $P(\neg A) \times (P(B \mid \neg A)^2 + P(\neg B \mid \neg A)^2)$ $-P(B)^2 - P(\neg B)^2$	$[-2, 2]$
M_{23}	Goodman and Kruskal	$\frac{\sum_i \max_j P(A_i B_j)+\sum_j \max_i P(A_i B_j)}{2-\max_i P(A_i)-\max_j P(B_j)}$ $-\frac{\max_i P(A_i)+\max_j P(B_j)}{2-\max_i P(A_i)-\max_j P(B_j)}$	$[0, +\infty)$
M_{24}	Normalized mutual information	$\frac{\sum_i \sum_j P(A_i B_j)\times\log_2 \frac{P(A_i B_j)}{P(A_i)P(B_j)}}{(-\sum_i P(A_i)\log_2 P(A_i))}$	$(-\infty, +\infty)$
M_{25}	J-Measure	$P(AB)\log\frac{P(B\|A)}{P(B)}$ $+P(A\neg B)\log\frac{P(\neg B\|A)}{P(\neg B)}$	$(-\infty, +\infty)$
M_{26}	One-way support	$P(B \mid A)\log_2\frac{P(AB)}{P(A)P(B)}$	$(-\infty, +\infty)$
M_{27}	Two-way support	$P(AB)\log_2\frac{P(AB)}{P(A)P(B)}$	$(-\infty, +\infty)$
M_{28}	Two-way support variation	$P(AB)\log_2\frac{P(AB)}{P(A)P(B)}$ $+P(A\neg B)\log_2\frac{P(A\neg B)}{P(A)P(\neg B)}$ $+P(\neg AB)\log_2\frac{P(\neg AB)}{P(\neg A)P(B)}$ $+P(\neg A\neg B)\log_2\frac{P(\neg A\neg B)}{P(\neg A)P(\neg B)}$	$(-\infty, +\infty)$
M_{29}	ϕ − Coefficient (Linear correlation coefficient)	$\frac{P(AB)-P(A)P(B)}{\sqrt{P(A)P(B)P(\neg A)P(\neg B)}}$	$(-\infty, +\infty)$
M_{30}	Piatetsky-Shapiro	$P(AB) - P(A)P(B)$	$[-1, 1]$
M_{31}	Cosine	$\frac{P(AB)}{\sqrt{P(A)P(B)}}$	$[0, +\infty)$
M_{32}	Loevinger	$1 - \frac{P(A)P(\neg B)}{P(A\neg B)}$	$(-\infty, 1]$
M_{33}	Information gain	$\log\frac{P(AB)}{P(A)P(B)}$	$(-\infty, +\infty)$
M_{34}	Sebag-Schoenauer	$\frac{P(AB)}{P(A\neg B)}$	$[0, +\infty)$
M_{35}	Least contradiction	$\frac{P(AB)-P(A\neg B)}{P(B)}$	$(-\infty, +\infty)$
M_{36}	Odd multiplier	$\frac{P(AB)P(\neg B)}{P(B)P(A\neg B)}$	$[0, +\infty)$
M_{37}	Example and counterexample rate	$1 - \frac{P(A\neg B)}{P(AB)}$	$(-\infty, 1]$
M_{38}	Zhang	$\frac{P(AB)-P(A)P(B)}{\max(P(AB)P(\neg B),P(B)P(A\neg B))}$	$(-\infty, +\infty)$

recall is defined as the probability of A given B (i.e., the proportion of instances where the postcondition A occurs among instances where the precondition B takes place).

We employ Le and Lo's mining algorithm (Le and Lo 2015) to estimate these probabilities (i.e., $P(A)$, $P(B)$, $P(AB)$, etc.) from execution traces. First, the algorithm splits the execution traces into sliding windows of size 5. Next, it computes the number of sliding windows where different conditions hold (see Table 4). Based on these numbers, we can then estimate various probabilities. For instances, $P(A) = \frac{N_w(A)}{N_w}$

Table 4 Notions for sliding window related counting

Notion	Description
$N_w(A)$	Number of sliding windows where A exists
$N_w(\neg A)$	Number of sliding windows where A does not exist
$N_w(B)$	Number of sliding windows where B exists
$N_w(\neg B)$	Number of sliding windows where B does not exist
$N_w(AB)$	Number of sliding windows where A is followed by B
$N_w(A\neg B)$	Number of sliding windows where A exists, but B does not exists + Number of sliding windows where both A and B exist, but A is not followed by B
$N_w(\neg AB)$	Number of sliding windows where A does not exist, but B exists
$N_w(\neg A\neg B)$	Number of sliding windows where A does not exist, and B does not exist
N_w	Total number of sliding windows

where $N_w(A)$ is the number of sliding windows where A exists and N_w is the total number of sliding windows.

3 Combining multiple metrics leveraging learning to rank

In this section, we first introduce the overall framework of this study and then elaborate the details of major components.

3.1 Overall framework

Figure 1 shows the framework of the proposed approach. It consists of two phases: training and deployment. In the training phase, our approach takes as input traces containing invocations of methods from classes whose specifications are known. The goal of the training phase is to learn a ranking model that composes the 38 measures such that correct rules can be ranked first before false positives (spurious rules). In the deployment phase, our approach takes as input traces containing invocations of methods from classes whose specifications are unknown. Our approach would then generate and rank rules using the ranking model learned in the training phase.

Four major components are part of the framework: candidate rule generator, score normalizer, learning to rank engine, and ranking model. We describe the functionality of each component as follows:

(1) Candidate rule generator This component runs Le and Lo's algorithm (Le and Lo 2015) to generate all possible temporal rules following the templates described

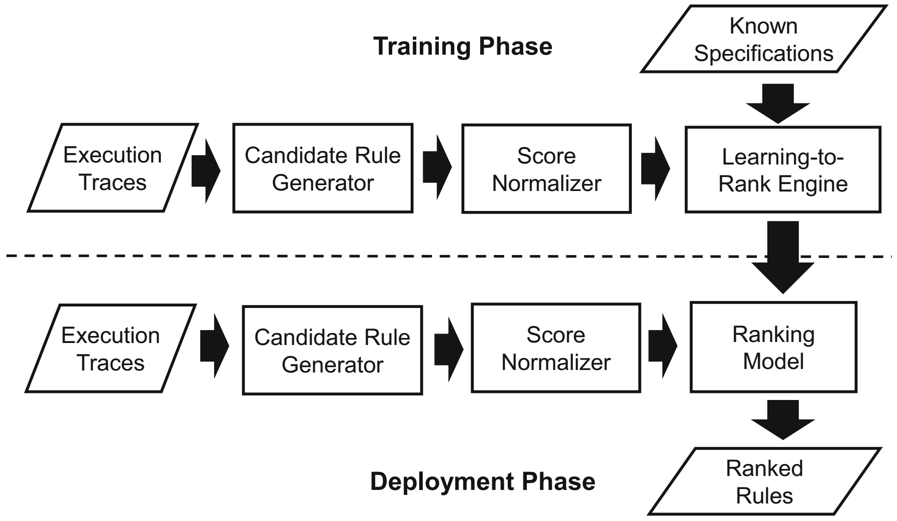


Fig. 1 Overall framework

in Sect. 2.1. For each temporal rule, it outputs 38 interestingness scores computed based on the input execution traces.

(2) Score normalizer Different interestingness measures have scores in different ranges. This component normalizes the scores of each measure so that they are in the same range, i.e., [0,1]. The normalized score S_norm_i of the ith measure is calculated as:

$$S\_norm_i = \frac{S_i - min\_S_i}{max\_S_i - min\_S_i}$$

In the above equation, S_i refers to the raw score of measure i. min_S_i and max_S_i represent the minimum and maximum value of S_i.

(3) Learning to rank engine This component takes as input a set of rules. For each rule, we have the scores of each of the 38 interestingness measures along with the ground truth label (i.e., whether it is a specification or not). Based on this input, it produces a ranking model that combines the different interestingness measures into one. Various machine learning algorithms can be used to produce this ranking model. We experiment with eight of them and describe their details in Sect. 3.2.

(4) Ranking model The ranking model is produced by the learning to rank engine. It takes as input a set of rules; for each rule, we have 38 scores corresponding to the 38 interestingness measures. It outputs a ranked list of rules computed based on the combination of the 38 scores.

3.2 Learning to rank algorithms

In the literature, many algorithms have been proposed to learn a good ranking model given a set of labeled data. These algorithms could be divided into three categories, i.e., point-wise, pair-wise, and list-wise, based on the optimization objective of the algorithm. Below we describe how each category of learning-to-rank algorithms works on rule-based specification mining task.

(1) Point-wise algorithms Point-wise algorithms treat ranking as a regular classification problem. In the context of rule-based specification mining, they create a model that can predict the correct label of each potential specification rule in the training data, with each rule being considered independently. The training loss that this type of algorithm minimize is:

$$L_{pointwise} = \sum_{i=1}^{N} loss(f(x_i), l_i)$$

where x_i is the feature vector representation one candidate rule, and l_i is the ground truth label, i.e., if the candidate rule is a specification, $l_i = 1$, otherwise $l_i = 0$. N is the total number of candidate rules in the training data. The function $loss(f(x_i), l_i)$ can be defined as the loss function of a classification problem or a regression problem.

(2) Pair-wise algorithms Pair-wise algorithms consider the order of rules in candidate rule pairs. Note that, candidate rule pairs are created within a set, in our case, a set means a class. The correct label of a candidate rule pair is a value representing the relative relevance of the two rules, e.g., whether one of them is more likely to be a specification than the other. Pair-wise approaches aim to learn a model that can minimize the number of pairs which are wrongly ordered (i.e., correct specifications are ordered after false positives). The candidate rules are then ranked based on the relative order predicted by the learned model. The training loss that this type of algorithm minimize is:

$$L_{pairwise} = \sum_{q=1}^{Q} loss(f(x_{qi}), f(x_{qj}), l_{qi}, l_{qj})$$

where q represents one target class (e.g., java.net.Socket). Q is the total number of considered classes. Function $loss(f(x_i), f(x_j), l_i, l_j)$ is a binary indicator function which tells whether the pair of candidate specification rules, i.e., x_{qi} and x_{qj}, are correctly ranked or not.

(3) List-wise algorithms List-wise algorithms are similar to pair-wise algorithms as they also consider the relationships among rules. However, different from pair-wise algorithms, they learn a model that minimizes a loss function which looks at the ordering of *all candidate rules* rather than just rule pairs. The training loss that this type of algorithm minimize is:

$$L_{listwise} = \sum_{q=1}^{Q} loss(f(x_{q1}), \ldots, f(x_{qn_q}), l_{q1}, \ldots, l_{qn_q})$$

where n_q means the number of candidate rules in target class q. Function $loss(f(x_{q1}), \ldots, f(x_{qn_q}), l_{q1}, \ldots, l_{qn_q})$ measures the degree of disagreement between the true ranked list of all documents associated with a query and the predicted ranked list of the same set of documents.

In this study, we consider the following eight learning to rank algorithms from the three categories.

Random forest (point-wise) Random forest is an ensemble learner[2] that creates many decision tree classifiers from subsets of training data and learns optimal weights of these trees to build a strong prediction model for unseen data (Breiman 2001). In software engineering, random forest has been widely applied and demonstrated its strong predictive power—c.f., (da Costa et al. 2014; Tian et al. 2015). The algorithm works in several steps. First draw a set of bootstrap samples from the training data. Then, for each of the samples, grow an unpruned classification or regression tree, with the following modification: at each node, rather than choosing the best split among all predictors, randomly sample some of the predictors and choose the best split from among those variables. In the testing phrase, the algorithm predicts the label of new data by aggregating the predictions of the sampled trees (i.e., majority votes for classification, average for regression).

MART (point-wise) MART is short for Multiple Additive Regression Trees, which is another ensemble learner built upon decision trees (Friedman 2001). Different from random forest that learns fully grown decision trees from subsets of data, MART combines weak decision trees, like shallow trees (i.e., low depth trees) to form a strong classifier.

RankNet (pair-wise) RankNet aims to minimize the number of inversions in ranking (Burges et al. 2005). An inversion means an incorrect order among a pair of results, i.e. when we rank a lower rated result above a higher rated result in a ranked list. RankNet uses a neural network combined with gradient descent steps to control the learning rate in each iteration step. The neural network has two hidden layers and uses backpropagation to minimize a cost function to perform the pairwise ranking.

RankBoost (pair-wise) RankBoost (Freund et al. 2003) is a boosting method that combines multiple weak rankers into a strong one. Boosting is a general technique for improving the accuracies of machine learning algorithms. The idea of boosting is to repeatedly construct "weak learners" by re-weighting training data, and to form an ensemble of weak learners such that the total performance of the ensemble is "boosted". RankBoost works in several iterations. At each iteration, a weak ranker is learned to optimize the ordering of rule pairs in the training data given their weights. The weight of a training rule pair corresponds to the importance of correctly predicting its label. Initially, the weights of all rule pairs are kept the same. At the end of each iteration, these weights are updated. Pairs that are incorrectly learned are emphasized in the next iteration by increasing their weights. The algorithm eventually produces

[2] Ensemble learners combine multiple learning algorithms to achieve higher classification accuracy.

a ranking model which is an ensemble of weak rankers, weighted based on their effectiveness in ranking rules in the training data.

Coordinate Ascent (list-wise) Coordinate Ascent is an algorithm that learns a ranking model, in the form of a linear combination of interestingness measures, which maximizes Mean Average Precision (MAP) when applied on the training data (Metzler and Croft 2007). MAP is a standard metric to evaluate the quality of a ranked list—its description is provided in Sect. 4.2. The value of MAP would be perfect (i.e., 1) if all specification rules are ranked higher than all non specification rules.

AdaRank (list-wise) AdaRank learns an ensemble of rankers which maximizes MAP when applied on the training data (Xu and Li 2007). AdaRank runs several rounds and at each round it creates a weak ranker and maintains a distribution of weights over the queries in the training data. Initially, AdaRank sets equal weights to the candidate rules. At each round, it increases the weights of those rules that are not ranked well by the current model. As a result, the learning at the next round will be focused on the creation of a weak ranker that can work on the ranking of those 'hard' rules. Finally, it outputs a ranking model by linearly combining the weak rankers.

ListNet (list-wise) ListNet is a version of RankNet which uses gradient descent to minimize a loss function given a ranking problem (Cao et al. 2007). Different from RankNet, ListNet optimizes directly for lists rather than rule pairs. ListNet implements the rank function as a neural network (NN), with the objective function set to be the cross entropy between two probability distributions over the object permutations, one derived from the human-labelled scores and the other derived from the model prediction (network output).

LambdaMART (both pair-wise and list-wise) LambdaMART (Wu et al. 2010) combines a tree-boosting algorithm MART with LambdaRank (Quoc and Le 2007), which is extended from RankNet. LambdaMART is one of the state-of-the-art learning to rank techniques and has been shown to be very successful in solving real world document retrieval problems (Svore et al. 2011).

4 Experiment

In this section, we first describe the evaluation dataset. Next, we introduce the metrics for evaluating the performance of a ranking approach. Subsequently, we describe our experiment setups. At the end of this section, we present three research questions and analyze experiments results by answering them.

4.1 Dataset

All experiments of this study are conducted on the benchmark dataset created by Le and Lo (2015). This data corpus contains manual identified specification rules for 28 classes from Java 6 SDK. These rules are inferred from the executions of 14 projects

Table 5 List of investigated classes from Java SDK

Class name	# Specification rules	# Candidate rules
java.net.Socket	19	240
java.net.URL	13	182
java.net.URLConnection	0	16
java.util.ArrayList	12	156
java.util.Collection	11	132
java.util.Deque	13	148
java.util.EnumMap	2	6
java.util.EnumSet	3	12
java.util.Formatter	2	6
java.util.HashMap	11	132
java.util.HashSet	10	110
java.util.Hashtable	13	168
java.util.IdentityHashMap	5	30
java.util.LinkedHashMap	11	132
java.util.LinkedHashSet	9	90
java.util.LinkedList	18	294
java.util.List	14	208
java.util.Map	11	132
java.util.NavigableMap	13	156
java.util.NavigableSet	9	90
java.util.Queue	9	72
java.util.Set	9	90
java.util.SortedMap	12	130
java.util.SortedSet	9	90
java.util.StringTokenizer	5	28
java.util.TreeMap	14	180
java.util.TreeSet	10	110
java.util.WeakHashMap	7	32
Total	274	3172

in DaCapo benchmark.[3] Table 5 summarizes the basic statistics of the dataset. In total, we consider 3,172 temporal rules with 274 of them being specifications.

4.2 Evaluation metrics

We consider three evaluation metrics: Mean Average Precision (MAP), Success@N, and Effort@P. Their definitions are given below.

[3] http://dacapobench.org/.

1. *Mean Average Precision (MAP)* MAP is a popular evaluation metric for evaluating a ranking approach. It is computed by taking the mean of the *average precision* of all ranked lists produced by an approach. In our setting, each ranker would produce a ranked list for each class that we consider. The average precision of a ranked list of rules produced for a given class is computed as:

$$AP = \frac{1}{total_s} \sum_{i=1}^{total_s} \frac{i}{Rank_i}$$

   In the above equation, $total_s$ refers to the total number of specification rules (i.e., true positives) in the ranked list, and $Rank_i$ is the position of the i-th specification rule in the ranked list. After calculating AP for each ranked list, the Mean Average Precision (MAP) is the mean of APs over all ranked lists, i.e.,

$$MAP = \frac{\sum_{i=1}^{N} AP}{N}$$

   In the above equation, N is the number of classes that are considered for evaluation.
2. *Success@N* Success@N is defined as the number of specification rules (i.e., true positives) found in the top-N rules returned by a ranking approach. This metric is preferred if developers only check a few possible specification rules provided by a tool. Success@N is referred to as Correct Rule@K in Le and Lo's paper (Le and Lo 2015), and it is also referred to as accuracy@N (Tamrawi et al. 2011), or Hit@N (Wang and Lo 2014).
3. *Effort@P* Effort@P is defined as the number of rules examined before finding P (e.g., 90%) of specification rules (i.e., true positives). This metric measures the amount of effort that needs to be spent on checking results returned by a ranking approach in order to retrieve a specific amount of specification rules.

4.3 Experiment settings

We use the implementations of the eight learning to rank algorithms in RankLib (Dang 2016). We modify RankLib source code to facilitate calculation of Effort@P, which is not implemented inside RankLib.

We perform K-fold cross validation to evaluate each ranker. The dataset is divided into K folds, and the whole evaluation process is repeated K times. For each time, one of the K folds is used as the test set while the other K-1 folds are combined to form a training set. In this study, the default value of K is set to 2, which means 50% of the data are used in the training phase (i.e., 14 out of the 28 classes) and the rest are used in the testing phase. To further reduce threats to validity, we run K-fold cross validation 10 times and calculate the average performance of all runs.

4.4 Research questions

In this study, we investigate the following three research questions.
(RQ1) How effective is our learning to rank based approach?

This question aims to investigate the performance of our learning to rank based approach compared to using a single interestingness measure. Considering that we have eight learning to rank algorithms to investigate, we answer RQ1 in two steps:

1. RQ1-a: Which learning to rank algorithm performs the best?
2. RQ1-b: Could the best performing learning to rank algorithm perform better than ranking based on a single interestingness measure?

In the first sub-question, i.e., RQ1-a, we investigate the performance of different learning to rank algorithms to determine which one is the most appropriate one for our problem. In the second sub-question, i.e., RQ1-b, we compare the performance of the best learning to rank based approach with that of ranking using a single interestingness measure.
(RQ2) What are the most important measures for the best performing ranking model?

Previously, Le and Lo shows that among the 38 possible interestingness measures, many of them are better than the frequently considered measures, i.e., support and confidence (Le and Lo 2015). However, when all the measures are considered together in our learning to rank based approach, it is unclear whether the most effective single measures are still important for identifying specification rules or not. Thus in this research question, we would like to know which measures contribute the most for the best performing ranking model.
(RQ3) How do different inputs and settings influence the performance of our approach?

In this research question, we investigate the impact of varying the amount of training data, and the need of performing score normalization. This question can be divided into two sub-questions:

1. RQ3-a: Does the amount of training data impact the performance of our model?
2. RQ3-b: Is normalization needed before training the model?

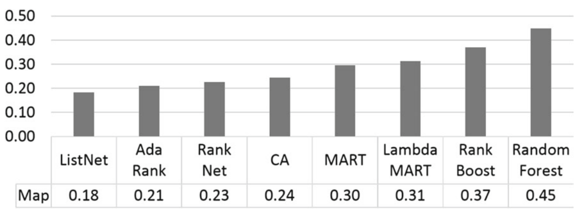
4.5 Experiment results

We present our experimental results as answers to the three research questions: RQ1-RQ3.

4.5.1 (RQ1) How effective is our learning to rank based approach?

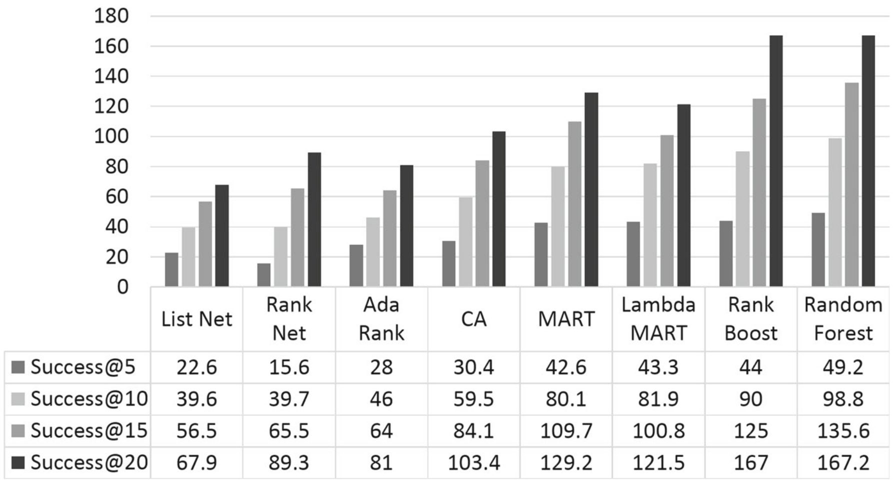
(RQ1-a) Which learning to rank algorithm performs the best?

Approach To answer this question, we compare the performance of the eight ranking algorithms introduced in Sect. 3, i.e., Random Forest, MART, RankNet, RankBoost, AdaRank, Coordinate Ascent, LambdaMART, and ListNet, on our dataset. To evaluate the performance of a ranking algorithm, we consider three metrics introduced in Sect. 3, i.e., Mean Average Precision (MAP), Success@N, and Effort@P. For Success@N



	ListNet	Ada Rank	Rank Net	CA	MART	Lambda MART	Rank Boost	Random Forest
Map	0.18	0.21	0.23	0.24	0.30	0.31	0.37	0.45

Fig. 2 Mean Average Precision of each ranking algorithm. CA is short for Coordinate Ascent algorithm



	List Net	Rank Net	Ada Rank	CA	MART	Lambda MART	Rank Boost	Random Forest
■ Success@5	22.6	15.6	28	30.4	42.6	43.3	44	49.2
■ Success@10	39.6	39.7	46	59.5	80.1	81.9	90	98.8
■ Success@15	56.5	65.5	64	84.1	109.7	100.8	125	135.6
■ Success@20	67.9	89.3	81	103.4	129.2	121.5	167	167.2

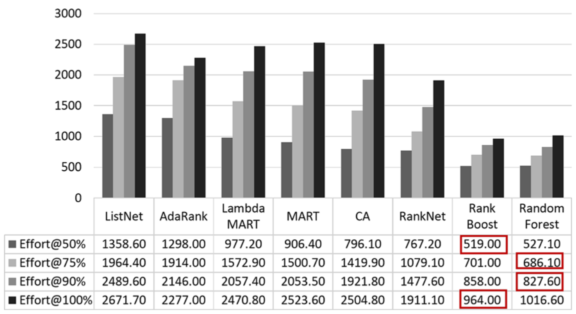
Fig. 3 Success@N of each ranking algorithm. CA is short for Coordinate Ascent algorithm. Note that we have 274 correct specifications in our dataset

metric, we consider the following values of K: 5, 10, 15, and 20. For Effort@P, we consider the following values of P: 50, 75, 90, and 100%.

Result Figure 2 shows the Mean Average Precision for the eight ranking algorithms. This table shows that the Random Forest ranking algorithm performs the best with a MAP of 0.45, followed by RankBoost and LambdaMART.

Figure 3 shows the Success@N of the eight ranking algorithms. We find that Random Forest, RankBoost, and LambdaMART performs better than the other algorithms for all considered K values. When K is set to 15 and 20, MART also performs well, even better than LambdaMART. From the results, one can find that when K is set to a small value, e.g., 5, the performance of the best ranking algorithms, e.g., MART, Lambda MART, RankBoost and Random Forest, do not differ much, however when K is increased to 15 and 20, the results are substantially different for these algorithms.

Figure 4 shows the Effort@P for the eight ranking algorithms. As the value of P increases, the number of rules needed to be examined increases too. Among the eight algorithms, Random Forest and RankBoost, consistently requires developers to spend less effort to identify 50, 75, 90, and 100% specification rules. RankBoost performs the best when P is set to 50 and 100%, while Random Forest performs the best when P is set to 75 and 90%.



	ListNet	AdaRank	Lambda MART	MART	CA	RankNet	Rank Boost	Random Forest
■ Effort@50%	1358.60	1298.00	977.20	906.40	796.10	767.20	519.00	527.10
■ Effort@75%	1964.40	1914.00	1572.90	1500.70	1419.90	1079.10	701.00	686.10
■ Effort@90%	2489.60	2146.00	2057.40	2053.50	1921.80	1477.60	858.00	827.60
■ Effort@100%	2671.70	2277.00	2470.80	2523.60	2504.80	1911.10	964.00	1016.60

Fig. 4 Effort@P of each ranking algorithm. CA is short for Coordinate Ascent algorithm. Note that we have 3172 candidate rules in our dataset

Table 6 Performances of learning to rank based on random forest and single measure approaches

Evaluation metric	Learning to rank (random forest)	Best single (measure name)	*p* value	Confidence	Support
MAP	0.45	0.27 (Odds ratio)	0.0009	0.24	0.23
Success@5	49.2	37 (Odds ratio)	0.0028	25	30
Success@10	98.8	64 (Odds ratio)	0.0028	44	54
Success@15	135.6	82 (Odds ratio)	0.0028	70	69
Success@20	167.2	102 (Odds ratio)	0.0029	95	90
Effort@50%	527.1	753 (Prevalence)	0.0029	772	991
Effort@75%	686.1	836 (Prevalence)	0.0010	1247	1493
Effort@90%	827.6	938 (Prevalence)	0.0029	1818	2202
Effort@100%	1016.6	1052 (Prevalence)	0.0029	2351	2668

p value is calculated by performing Wilcoxon Signed-Rank Test

Overall, Random Forest in general outperforms the other seven ranking algorithms considering the three metrics.

(RQ1-b) Could the best performing learning to rank algorithm performs better than ranking based on any single interestingness measure?

Approach To answer this research question, we compare the effectiveness of the best learning to rank based model (i.e., Random Forest based) with the performance of using any single interestingness measure. Similar to RQ1-a, we consider MAP, Success@5, 10, 15, 20 and Effort@50%, 75, 90, and 100%.

Result Table 6 shows the comparison of our learning to rank approach (based on Random Forest) and the approaches based on a single measure. Besides the performance of the best single measure based approach, we also present the performances of two commonly used single measures, i.e., Confidence and Support in Table 6. The experiment results show that among all the single measure based approaches, Odds Ratio has the highest MAP score and Success@N scores, while Prevalence has the lowest Effort@P scores. Compared to the best single measure, our approach can achieve a much better performance, i.e., 35.41–66.22% improvement in terms of MAP and Success@N. We also find that our learning to rank based approach can reduce 3.37–30% of the effort needed to identify the same number of specification rules. We also perform

Wilcoxon signed-rank test (Wilcoxon 1945) based on 10 runs results. For MAP and Success@N metrics, we test the alternative hypothesis: "The scores of our learning-to-rank based approach are greater than the best single measure based approach". For Effort@P, we test the alternative hypothesis: "The scores of our learning-to-rank based approach are less than the best single measure based approach". The p values produced by our Wilcoxon signed-rank test are included in Table 6. Considering a commonly-used significance-level of 0.05, the p values show that our approach statistically significantly outperforms the best single measure approach.

Table 6 also shows that in term of Success@5, the benefit of applying learning to rank approach is not as high as the other Success@N, which might suggest that for all ranking approaches, getting a very high Success@5 score is very hard. Similarly when the value of P in Effort@P increases from 90 to 100%, the benefit of using learning to rank drops rapidly, which suggests that some specifications are hard to identify by all approaches.

Learning to rank approach based on Random Forest generally performs the best among the other alternative ranking algorithms. It can improve the MAP and Success@N of ranking using a single measure by 35.41%-66.22% and reduce Effort@P by 3.37%-30.00%.

4.5.2 *(RQ2) What are the most important measures for the best performing ranking model?*

Approach For Random Forest, the output learned model is a forest made of many decision trees. To compute the importance of each measure, we consider *Mean Decrease Gini* ($I_G(Measure)$), which measures how each variable contributes to the homogeneity of the nodes and leaves in the resulting forest (Louppe et al. 2013).

Given a decision tree, which is the basic unit of the learned Random Forest model, the Gini impurity, i.e., $G(i)$ of a splitting node i is defined as:

$$G(i) = 1 - {p_1}^2 - {p_0}^2$$

In the above equation, $p_k(k = 0, 1)$ is defined as n_k/n, where 0 and 1 refers to non-specification rules (false positives) and specification rules (true positives) respectively. n_k is the number of rules at node i with class k, and n is the total number of rules at node i. For example, if at a decision node, 10 rules reach this node following the decision tree, and 3 of them are specification rules and the other 7 are non-specification rules, then the Gini impurity of this node is $1 - 0.3^2 - 0.7^2 = 0.42$.

The decrease of Gini impurity, i.e., $\delta G(i)$ at the splitting node i is then defined as:

$$\delta G(i) = G(i) - p\_left * G(i\_left) - p\_right * G(i\_right)$$

In the above equation, p_left and p_right represent the percentage of data split to the left/right side child of the current node. For example given the current node with metric support and threshold 10, rules satisfying $support < 10$ should go to the left side of this splitting node; similarly, rules satisfying $support > 10$ should go to the right side of this node. $G(i\_left)$ and $G(i\_right)$ refer to the Gini impurity of the left child node and right child node of the current splitting node.

Finally, the Mean Decrease Gini of an interestingness measure, i.e., $I_G(Measure)$, is computed by traversing all the decision trees built from training data and averaging sum of the weighted $\delta G(i)$ for every node related to the measure:

$$I_G(Measure) = \frac{1}{M}\sum_m \sum_i \frac{N_i}{N}\delta G(i)$$

In the above equation, M is the total number of trees in the forest, where $\frac{N_i}{N}$ refers to the percentage of rules at node i.

Result We compute the average $I_G(Measure)$ for the 38 interesting measures over the five runs of our learning to rank based approach. After each run, we rank the 38 measures based on their Mean Decrease Gini importance scores in descending order, i.e., the most important metric has a rank of 1. The average rank of each single metric is shown in Table 7. This figure shows that Prevalence is the most important measure for the Random Forest ranking model, followed by *Certainty Factor* and *Specificity*.

Prevalence (M4) is the most importance interestingness measure for the Random Forest based learning to rank approach, followed by *Certainty Factor* (M13) and *Specificity* (M6).

4.5.3 (RQ3) How do different inputs and settings influence the performance of our approach?

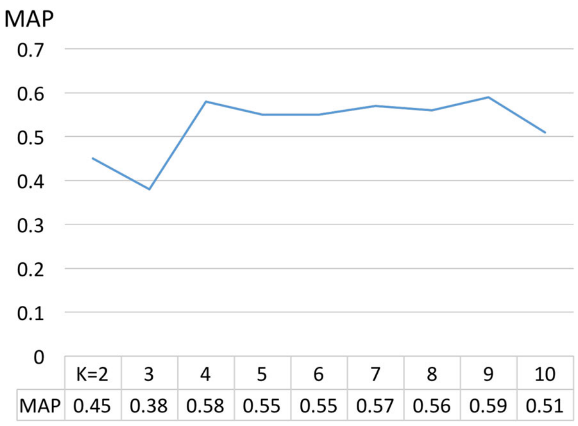
(RQ3-a) Does the amount of training data impact the performance of our model?

Approach To answer this research question, we vary the K of K-fold cross validation from 2 to 10, and investigate how the variation of K impacts the performance of our approach. With a larger value of K, we have more training data.

Result Figures 5, 6 and 7 show how Random Forest learning to rank approach performs when we increase the value of K. We find that results of all metrics are mostly consistent when K varies, i.e., when K = 4–9, performance is stable and better than when K = 2,3, and 10 for all three metrics. We also perform Wilcoxon signed-rank test (Wilcoxon 1945) based on 10 runs results, to compare the results for K = 2,3 and 10 with the best results (K = 9). We consider 3 evaluation metrics, i.e., MAP, Success@10, and Effort@90%. For MAP and Success@10 metrics, we test the alternative hypothesis: "The scores of our model under 9-fold setting are greater than our model under 2/3/10-fold setting". For Effort@P, we test the alternative hypothesis: "The scores of our model under 9-fold setting are less than our model under 2/3/10-fold setting". The p values produced by our Wilcoxon signed-rank test are shown in Table 8. Considering a commonly-used significance-level of 0.05, the p values show that the model when

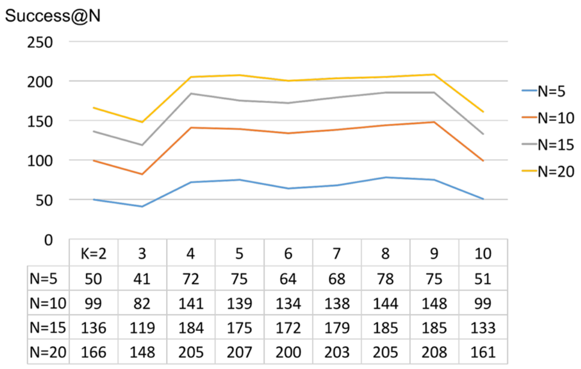
Table 7 Importance of each single metric for learning to rank approach based on Random Forest

Final rank	Measure name	Average rank
1	M4: Prevalence	1.0
2	M13: Certainty factor	4.3
3	M6: Specificity	6.4
4	M33: Information gain	7.4
5	M7: Accuracy	7.6
6	M32: Loevinger	8.0
7	M15: Yule'Q	8.6
8	M37: Example and counterexample rate	9.9
9	M2: Confidence	11.9
10	M20: Collective strength	12.2
11	M23: Goodman and Kruskal	12.3
12	M38: Zhang	13.3
13	M14: OddsRatio	13.4
14	M21: Laplace correction	14.3
15	M10: Added value/change of support	15.1
16	M16: Yule'Y	15.5
17	M35: Least contradiction	16.6
18	M8: Lift/interest	18.7
19	M5: Recall	18.7
20	M9: Leverage	19.4
21	M24: Normalized mutual information	20.7
22	M34: Sebag-Schoenauer	21.4
23	M26: One-WaySupport	23.1
24	M29: $\phi -$ Coefficient (Linear correlation coefficient)	23.2
25	M31: Cosine	23.3
26	M36: Odd multiplier	23.8
27	M18: Conviction	24.4
28	M3: Coverage	24.8
29	M11: Relative risk	28.5
30	M27: Two-way support	29
31	M12: Jaccard	30.9
32	M28: Two-way support variation	31.5
33	M22: Gini index	31.7
34	M19: Interestingness weighting dependency	33.4
35	M30: Piatesky-Shapiro	33.9
36	M1: Support	33.9
37	M25: J-Measure	34
38	M17: Klosgen	34.7



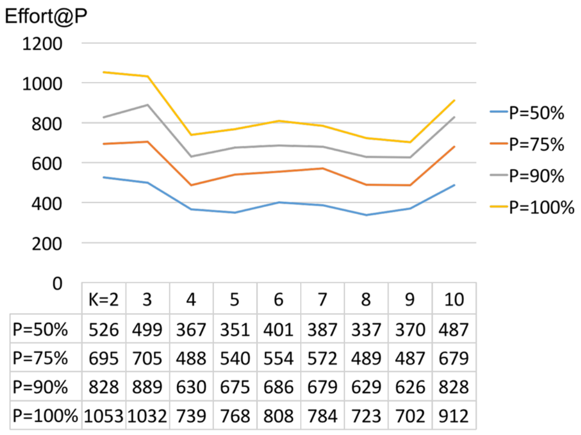
	K=2	3	4	5	6	7	8	9	10
MAP	0.45	0.38	0.58	0.55	0.55	0.57	0.56	0.59	0.51

Fig. 5 Mean Average Precision (MAP) of our approach when varying the value of K for K-Fold cross validation



	K=2	3	4	5	6	7	8	9	10
N=5	50	41	72	75	64	68	78	75	51
N=10	99	82	141	139	134	138	144	148	99
N=15	136	119	184	175	172	179	185	185	133
N=20	166	148	205	207	200	203	205	208	161

Fig. 6 Number of specification rules found in Top-N rules (Success@N) of our approach when varying the value of K for K-Fold cross validation



	K=2	3	4	5	6	7	8	9	10
P=50%	526	499	367	351	401	387	337	370	487
P=75%	695	705	488	540	554	572	489	487	679
P=90%	828	889	630	675	686	679	629	626	828
P=100%	1053	1032	739	768	808	784	723	702	912

Fig. 7 Number of rules examined before finding P of specification rules (Effort@P) of our approach when varying the value of K for K-Fold cross validation

Table 8 *p* Value of Wilcoxon signed-rank test when comparing K = 2,3,10 with K = 9 in K-fold cross validation

	MAP	Success@10	Effort@90%
K = 2 versus K = 9	0.00097	0.00097	0.00097
K = 3 versus K = 9	0.00289	0.00288	0.00286
K = 10 versus K = 9	0.00097	0.00097	0.00097

Table 9 Performance with and without normalization

	MAP	Success@10	Effort@90%
Without norm	0.41593	90.7	1098.9
With norm	0.45	98.8	827.6
Difference	7.42%	6.28%	23.83%

K = 9 statistically significantly outperforms the model when K = 2, 3, 10. One potential reason to explain why the results for K = 4–9 are better than those for K = 2,3 is that the training data may not be enough when K = 2 and 3. And there might be an overfitting problem when K = 10.

(RQ3-b) Is normalization needed before training the model?

Approach To answer this question, we compare the performance of Random Forest based learning to rank approach with and without score normalization.

Result Table 9 shows the performance of our approach with and without normalization. It shows that for MAP and Success@10, normalization can improve the performance by 7.42 and 6.28% respectively. It can also reduce the amount of effort needed to identify 90% specification rules by 23.83%.

The performance of our best performing learning to rank algorithm (i.e., Random Forest) remains stable for different amount of training data (i.e., K = 4-9). Moreover, the performance of our proposed approach is boosted when normalization is employed.

5 Discussion

In this section, we first discuss how will our approach perform when only considering the most important features. Next, we discuss an alternative to combine multiple measures together, i.e., the unsupervised data fusion technique. In the end, we discuss several threats to validity of this study.

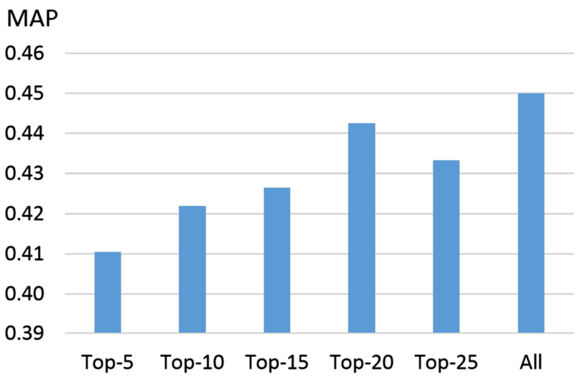


Fig. 8 MAP when varying the number of selected features

5.1 Learning-to-rank only using important measures

In Sect. 4.5.2, we have analyzed which features (i.e., interesting measures) are important to the best performing ranking models. However, it is unknown whether leveraging the top-N most important features can result in construction of better ranking models, compared to using all features. Thus in this subsection, we take top-5, 10, 15, 20, and 25 most important measures (shown in Table 7) as input features for training ranking models. The results are shown in Figs. 8, 9 and 10. From the figures, we can note that top-5 and top-10 features result in less effective models than the one trained by all features in terms of MAP, Success@10, and Effort@90%. One possible reason might be only using up to 10 interestingness measures could not cover all the major information carried by the 38 interestingness measures. As we consider more interesting measures, we find that (1) using all 38 metrics results in the best performing ranking model in terms of MAP scores; (2) using top-15 and top-20 features give better results than using all features in terms of Success@10; (3) using top-20 and top-25 features give better results than using all features in terms of Effort@90%. Since there are no consistent patterns for the best selection of top features to construct effective ranking models, we suggest practitioners to train and test our proposed models on a fraction of their own dataset to determine which set of features is best for training good models for their data.

5.2 Alternative: unsupervised data fusion

In this paper, we consider a supervised machine learning technique, i.e., learning-to-rank, to identify specification rules. As an alternative, one may choose to employ unsupervised data fusion (Wu 2012) to combine the 38 interestingness measures. To investigate the performance of using unsupervised machine learning techniques for combining multiple interestingness measures, we did an initial experiment using the same dataset as Sect. 4. Our initial investigation finds that a composite measure created using data fusion technique is unable to outperform the best performing single measure, not to mention our new learning to rank based approach.

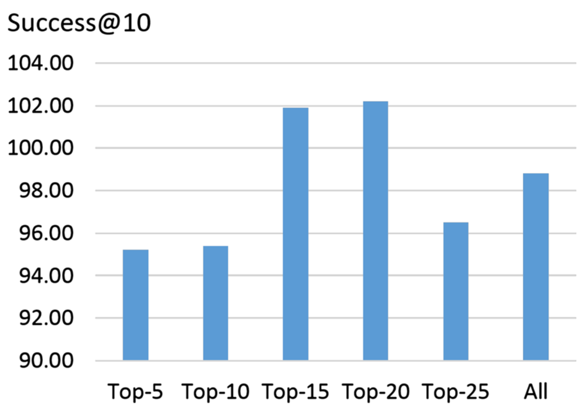


Fig. 9 Success@10 when varying the number of selected features

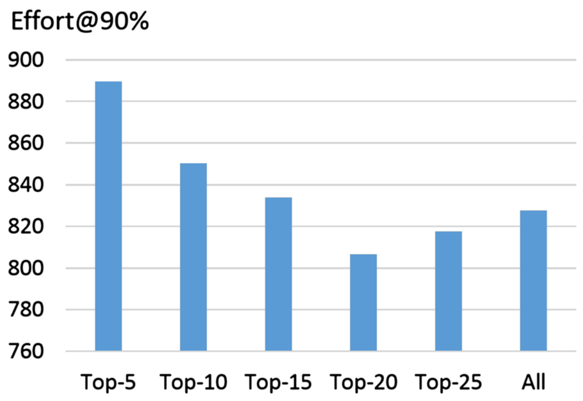


Fig. 10 Effort@90% when varying the number of selected features

In our initial experiment, we have implemented five data fusion algorithms, including CombSUM (Fox et al. 1993), CombMNZ (Fox et al. 1993), CombANZ (Fox et al. 1993), and two correlation-based methods corrA and corrB (Wu 2012). The first three algorithms simply combine the scores from different measures by summing up all values of measures, computing the average of the non-zero values, multiplying the summation of all values with the number of measures that assign a non-zero value to a rule, respectively. The last two algorithms first select a set of measures based on the degree of overlap of their returned ranked list and combine the values provided by selected measures. Due to page limit, for a detailed description, please refer to Lucia et al.'s work which applies data fusion techniques to combine interestingness measures for locating faults (Lo et al. 2014).

Table 10 presents the results of the five data fusion approaches. It shows that unsupervised data fusion approaches perform worse than the best single measure for all considered metrics. A reason for the bad performance might be that some non-relevant interestingness measures are selected and considered equally with relevant interestingness measures.

Table 10 Data fusion approaches versus single measure approach

Evaluation metric	Algorithm	Data fusion performance	Best single (measure name)	Difference (%)
MAP	CombSUM	0.22	0.27 (Odds ratio)	−17.41
	CombANZ	0.16		– 40.34
	CombMNZ	0.23		– 14.07
	CorrA	0.23		– 14.33
	CorrB	0.23		– 15.54
Success@10	CombSUM	45	64 (Odds ratio)	– 29.69
	CombANZ	29		– 54.69
	CombMNZ	46		– 28.13
	CorrA	49		– 23.44
	CorrB	44		– 31.25
Effort@90%	CombSUM	2671	938 (Prevalence)	– 184.75
	CombANZ	2674		– 185.07
	CombMNZ	2271		– 142.11
	CorrA	2598		– 176.97
	CorrB	2677		– 185.39

5.3 Threats to validity

Threats to construct validity refers to the suitability of our evaluation metrics. In this study, to reduce threats introduced by evaluation metrics, we have considered three general evaluation metrics for ranking problem, i.e., MAP, Success@N, and Effort@P. These metrics have been used in many prior software engineering works to measure the effectiveness of a ranking approach (Liu 2011).

Threats to internal validity refers to potential errors in our experiments. The implementation of the learning to rank algorithms are taken from a mature library RankLib, which has been integrated into the well-known Lemur open source project. Additionally, we have checked our code, but there might still be errors that we did not notice. To minimize a possible bias due to the randomness involved in splitting the data into K folds, following the advice made by Arcuri and Briand (2014) to evaluate algorithms involving a random process, we run K-fold cross validation 10 times. A similar strategy was also performed by a number of prior work (Zhang et al. 2016; Ghotra et al. 2017). We have also checked the stability of results in the 10 runs. We have found that the 10 runs results do not different much, i.e., the performance of our approach under a fixed K-fold setting is stable. For instance, the ten Success@10 scores when we perform 2-fold cross validation with Random Forest are: 103, 97, 96, 99, 99, 101, 99, 98, 99, and 97 (with an average of 98.8, as shown in Fig. 3).

Threats to external validity refers to the generalizability of our findings. In our experiments, we consider the same benchmark dataset considered by Le and Lo (2015), which contains specification rules of classes from Java 6 SDK. We plan to manually label more specification rules and test our approach on a larger dataset in the future.

In this work, we evaluated and compared our approach with baselines on mining two-events temporal rules, i.e., method call A is always followed by B and method call A is always preceded by B. We focus on two-events rules because this type of rule is mostly considered/used as specification rules (Yang et al. 2006). Additionally, most complex rules can be composed by a set of simpler rules. In the future, we plan to apply our approach for mining more types of rules.

6 Related work

In this section, we first introduce several work on specification mining. Next, we introduce some applications of learning to rank technique in the Software Engineering area.

6.1 Specification mining

There are several existing approaches that infer rules from program execution traces. Yang et al. propose Perracotta that mines two-event temporal rules from execution traces. To infer these rules, Perracotta uses a set of predefined rule templates and partitions input traces to several sub-traces. It computes satisfaction rate of a template, which is the number of partitions satisfying the template divided by the number of total partitions (Yang et al. 2006). Lo et al. extend Yang et al.'s approach by inferring from execution traces temporal rules with arbitrary lengths instead of two-event rules (Lo et al. 2008). Li et al. also extend Yang et al.'s work by extracting simple linear temporal logic (LTL) rules from execution traces for hardware design (Li et al. 2010). Gruska et al. extract temporal properties of API usage and employ these properties to detect anomalies that deviate from the 6000 projects (Gruska et al. 2010). Lo et al. mine length-2 quantified temporal rules which specify data-flow dependency constraints between method invocations (Lo et al. 2012). Lo et al. also infer rules following the concept of Live Sequence Charts (LCSs), which are enriched with Daikon-style constraints (Lo and Maoz 2012). Le and Lo investigate the effectiveness of several interestingness measures from data mining community for inferring rule-based specifications. Their findings indicate that other measures besides support and confidence can better detect correct two-event temporal rules from execution traces (Le and Lo 2015). Lemieux et al. introduce Texada that mines temporal specifications in the form of linear temporal logic (LTL) of arbitrary length and complexity (Lemieux et al. 2015). Compared to existing rule-based specification mining studies, our work utilizes 38 interestingness measures and leverages learning to rank algorithms to differentiate specification rules (i.e., true positives) from spurious ones (i.e., false positives).

There are other existing studies that mine specifications in other formats. Lo et al. propose SMArTIC that applies a variant of k-tails automaton learning algorithm (Biermann and Feldman 1972) to infer finite state automatons (FSAs) from a set of execution traces (Lo and Khoo 2006). Walkinshaw et al. propose FSA inference framework that enables developers to add their LTL constraints to reduce minimum amount of input traces to mine reasonably precise specifications (Walkinshaw and Bogdanov 2008). The Synoptic tool by Beschastnikh et al. automatically mines three types of temporal rules from execution traces and uses them to generate a concise finite state automaton (FSA) model that satisfies these rules (Beschastnikh et al. 2011). Krka et al. propose several algorithms that mine FSA based specifications from execution traces and likely invariants inferred by Daikon (Krka et al. 2014). Le et al. propose SpecForge that synergizes different FSA-based specification miners by introducing novel concepts of model fission and model fusion (Le et al. 2015). Walkinshaw et al. employs data mining classifiers in their proposed inference algorithm that mines Extended Finite State Machines (EFSMs) from execution traces (Walkinshaw et al. 2016). Different from the above studies, we infer specifications in forms of temporal rules rather than finite state automata.

6.2 Learning to rank applications in software engineering

Learning to rank algorithms have been applied to several software engineering research studies. Xuan et al. propose Multric that employs RankBoost algorithm (Freund et al. 2003) to combine scores computed by 25 spectrum-based fault localization formulas (Xuan and Monperrus 2014). Le et al. propose Savant that utilizes rankSVM algorithm (Lee and Lin 2014) and likely invariants inferred by Daikon (Ernst et al. 2007) to localize faults (Le et al. 2016). Le et al. (2016)'s empirical evaluation indicates that Savant outperforms many state-of-the-art spectrum-based fault localization approaches including Multric (Xuan and Monperrus 2014). Ye et al. propose a learning to rank approach for information retrieval (IR) based bug localization using features extracted from textual bug reports and source code files (Ye et al. 2014). Yang et al. employs learning to rank and feature selection for software defect prediction (Yang et al. 2015). Learning to rank algorithms are also utilized to improve the effectiveness of information retrieval applications in software engineering. Binkley et al. apply learning to rank algorithms to improve several feature location models for software maintenance (Binkley and Lawrie 2014). Niu et al. propose a code example search approach that employs a learning to rank technique to train a ranking schema (Niu et al. 2016). Zhou et al. design BugSim, a learning to rank based method to detect duplicate bug reports from a dataset of 45,100 real bug reports of twelve Eclipse projects (Zhou and Zhang 2012). Yuan et al. propose a learning to rank based model that leverages information from both developers' activities and output of bug report localization task for bug report assignee recommendation (Tian et al. 2016). Learning to rank algorithms have not been employed to mine specifications. Our study is the first to explore the effectiveness of learning to rank solutions for rule-based specification mining.

7 Conclusion and future work

Rule-based specification mining focuses on mining specifications as a set of rules. Current state-of-the-art rule-based specification mining approaches rely on interesting measures, e.g., support and confidence, to predict the probability of a candidate rule to be a true specification. However, a recent study by Le and Lo found that among 38 interestingness measures, the most frequently considered measures, i.e., support and confidence, could not identify more specification rules than some of the other interesting measure like Odds Ratio. Their finding suggests that developers should carefully pick interestingness measure for mining specification rules.

In this paper, we extend the work of Le and Lo by proposing a learning to rank based approach to consider all the 38 available interestingness measures together and investigate whether such combination helps or not. Our learning to rank approach takes a set of known specification rules as input and learns the best combination over the 38 measures using random forest ranking algorithm. The experiment results for classes from Java 6 SDK show that our learning to rank based approach can improve the best performance of ranking using a single measure by up to 66%. We also find that Prevalence, Certainty Factor, and Specificity are the most important measures for our learning to rank based approach.

In the future, we plan to test our learning to rank approach by considering more classes and libraries, and propose additional measures and strategies to better differentiate specification rules from spurious ones. We would also like to investigate effectiveness of techniques such as principal component analysis to reduce the feature dimension for our approach. Furthermore, we plan to extend our approach further to infer 3-event temporal rules. The space of possible 3-event temporal rules is larger than that of 2-event ones, which poses accuracy and scalability challenges. It would be harder to infer specification rules since there is a need to sieve them out from a larger number of non-specification ones. The exploration of the larger search space also implies a higher computational cost.

References

Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. **24**(3), 219–250 (2014)

Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 267–277. ACM (2011)

Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. **100**(6), 592–597 (1972)

Binkley, D., Lawrie, D.: Learning to rank improves IR in SE. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 441–445. IEEE Computer Society (2014)

Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. In: Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22–26, 2006, Portland, Oregon, USA, pp. 169–190 (2006)

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

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.: Learning to rank using gradient descent. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 89–96. ACM (2005)

Cao, Z., Qin, T., Liu, T.Y., Tsai, M.F., Li, H.: Learning to rank: from pairwise approach to listwise approach. In: Proceedings of the 24th International Conference on Machine Learning, pp. 129–136. ACM (2007)

da Costa, D.A., Abebe, S.L., McIntosh, S., Kulesza, U., Hassan, A.E.: An empirical study of delays in the integration of addressed issues. In: ICSME, pp. 281–290 (2014)

Dang, V.: Ranklib. https://sourceforge.net/p/lemur/wiki/RankLib/ (2016). Accessed 17 Sept 2016

Demsky, B., Ernst, M.D., Guo, P.J., McCamant, S., Perkins, J.H., Rinard, M.: Inference and enforcement of data structure consistency specifications. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 233–244. ACM (2006)

Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16–22, 1999, pp. 411–420 (1999)

Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)

Fahland, D., Lo, D., Maoz, S.: Mining branching-time scenarios. In: ASE (2013)

Fox, E.A., Koushik, M.P., Shaw, J., Modlin, R., Rao, D., et al.: Combining evidence from multiple searches. In: The First Text Retrieval Conference (TREC-1), US Department of Commerce, National Institute of Standards and Technology, vol. 500 (1993)

Freund, Y., Iyer, R., Schapire, R.E., Singer, Y.: An efficient boosting algorithm for combining preferences. J. Mach. Learn. Res. **4**(Nov), 933–969 (2003)

Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Ann. Stat. **29**, 1189–1232 (2001)

Gabel, M., Su, Z.: Online inference and enforcement of temporal properties. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Vol. 1, pp. 15–24. ACM (2010)

Geng, L., Hamilton, H.J.: Interestingness measures for data mining: a survey. ACM Comput. Surv. (CSUR) **38**(3), 9 (2006)

Ghotra, B., Mcintosh, S., Hassan, A.E.: A large-scale study of the impact of feature selection techniques on defect classification models. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 146–157. IEEE Press (2017)

Gruska, N., Wasylkowski, A., Zeller, A.: Learning from 6, 000 projects: lightweight cross-project anomaly detection. In: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 119–130 (2010)

Henning, J., Pfeiffer, D.U., et al.: Risk factors and characteristics of H5N1 highly pathogenic avian influenza (HPAI) post-vaccination outbreaks. Vet. Res. **40**(3) (2009). https://www.vetres.org/articles/vetres/abs/2009/03/v09120/v09120.html

Knight, J.C., DeJong, C.L., Gibble, M.S., Nakano, L.G.: Why are formal methods not used more widely? In: Fourth NASA Formal Methods Workshop, pp. 1–12 (1997)

Krka, I., Brun, Y., Medvidovic, N.: Automatic mining of specifications from invocation traces and method invariants. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 178–189. ACM (2014)

Le, T.D.B., Lo, D.: Beyond support and confidence: exploring interestingness measures for rule-based specification mining. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 331–340. IEEE (2015)

Le, T.D.B., Le, X.B.D., Lo, D., Beschastnikh, I.: Synergizing specification miners through model fissions and fusions (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 115–125. IEEE (2015)

Le, T.D., Lo, D., Le Goues, C., Grunske, L.: A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 177–188. ACM (2016)

Lee, C.P., Lin, C.J.: Large-scale linear ranksvm. Neural Comput. **26**(4), 781–817 (2014)

Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 81–92. IEEE (2015)

Li, Z., Zhou, Y.: Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In: ESEC/SIGSOFT FSE (2005)

Li, W., Forin, A., Seshia, S.A.: Scalable specification mining for verification and diagnosis. In: Proceedings of the 47th Design Automation Conference, pp. 755–760. ACM (2010)

Liu, T.Y.: Learning to rank for information retrieval. Found. Trends Inf. Retr. **3**(3), 225–331 (2009)

Liu, T.Y.: Learning to Rank for Information Retrieval. Springer, Berlin (2011)

Lo, D., Khoo, S.C.: Smartic: towards building an accurate, robust and scalable specification miner. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 265–275. ACM (2006)

Lo, D., Maoz, S.: Scenario-based and value-based specification mining: better together. Autom. Softw. Eng. **19**(4), 423–458 (2012)

Lo, D., Khoo, S.C., Liu, C.: Mining temporal rules for software maintenance. J. Softw. Maint. **20**(4), 227–247 (2008)

Lo, D., Ramalingam, G., Ranganath, V.P., Vaswani, K.: Mining quantified temporal rules: formalism, algorithms, and evaluation. Sci. Comput. Program. **77**, 743–759 (2012)

Lo, D., Xia, X., et al.: Fusion fault localizers. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 127–138. ACM (2014)

Louppe, G., Wehenkel, L., Sutera, A., Geurts, P.: Understanding variable importances in forests of randomized trees. In: Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K. Q. (eds.) Advances in Neural Information Processing Systems, pp. 431–439. Curran Associates, Inc. (2013). http://papers.nips.cc/paper/4928-understanding-variable-importances-in-forests-of-randomized-trees.pdf

Metzler, D., Croft, W.B.: Linear feature-based models for information retrieval. Inf. Retr. **10**(3), 257–274 (2007)

Microsoft.: Rules for WDM drivers. http://msdn.microsoft.com/en-us/library/windows/hardware/ff551714(v=vs.85).aspx. Accessed 18 Oct 2016 (2016)

Mutegi, C., Ngugi, H., Hendriks, S., Jones, R.: Prevalence and factors associated with aflatoxin contamination of peanuts from Western Kenya. Int. J. Food Microbiol. **130**(1), 27–34 (2009)

Niu, H., Keivanloo, I., Zou, Y.: Learning to rank code examples for code search engines. Empir. Softw. Eng. **22**(1), 259–291 (2016)

Quoc, C., Le, V.: Learning to rank with nonsmooth cost functions. Proc. Adv. Neural Inf. Process. Syst. **19**, 193–200 (2007)

Rothman, K.J.: Epidemiology: An Introduction. Oxford university press, Oxford (2012)

Safyallah, H., Sartipi, K.: Dynamic analysis of software systems using execution pattern mining. In: 14th International Conference on Program Comprehension (ICPC 2006), 14–16 June 2006, pp. 84–88. Greece, Athens (2006)

Stampfer, M.J.: Welding occupations and mortality from Parkinson's disease and other neurodegenerative diseases among united states men, 1985–1999. J. Occup. Environ. Hyg. **6**, 267–272 (2009)

Svore, K.M., Volkovs, M.N., Burges, C.J.: Learning to rank with multiple objective functions. In: Proceedings of the 20th International Conference on World Wide Web, pp. 367–376. ACM (2011)

Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 365–375. ACM (2011)

Tian, Y., Nagappan, M., Lo, D., Hassan, A.E.: What are the characteristics of high-rated apps? A case study on free android applications. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 301–310. IEEE (2015)

Tian, Y., Wijedasa, D., Lo, D., Le Gouesy, C.: Learning to rank for bug report assignee recommendation. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp 1–10. IEEE (2016)

Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 248–257. IEEE Computer Society (2008)

Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empir. Softw. Eng. **21**(3), 811–853 (2016)

Wang, S., Lo, D.: Version history, similar report, and structure: putting them together for improved bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, pp. 53–63. ACM (2014)

Wilcoxon, F.: Individual comparisons by ranking methods. Biom. Bull. **1**(6), 80–83 (1945)

Wu, S.: Data Fusion in Information Retrieval, vol. 13. Springer, Berlin (2012)

Wu, Q., Burges, C.J., Svore, K.M., Gao, J.: Adapting boosting for information retrieval measures. Inf. Retr. **13**(3), 254–270 (2010)

Xu, J., Li, H.: Adarank: a boosting algorithm for information retrieval. In: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 391–398. ACM (2007)

Xuan, J., Monperrus, M.: Learning to combine multiple ranking metrics for fault localization. In: ICSME-30th International Conference on Software Maintenance and Evolution (2014)

Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the 28th International Conference on Software Engineering, pp. 282–291. ACM (2006)

Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In: European Symposium on Research in Computer Security, pp. 163–182. Springer (2014)

Yang, X., Tang, K., Yao, X.: A learning-to-rank approach to software defect prediction. IEEE Trans. Reliab. **64**(1), 234–246 (2015)

Ye, X., Bunescu, R., Liu, C.: Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 689–699. ACM (2014)

Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E.: Cross-project defect prediction using a connectivity-based unsupervised classifier. In: Proceedings of the 38th International Conference on Software Engineering, pp. 309–320. ACM (2016)

Zhong, H., Su, Z.: Detecting API documentation errors. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2013, Part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013, pp. 803–816 (2013)

Zhou, J., Zhang, H.: Learning to rank duplicate bug reports. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, pp. 852–861. ACM (2012)