Finding Relevant Files for Bug Reports Based on Mean Reciprocal Rank Maximization Approach

Duc-Trong Le
DS&KTLab, University of Engineering and Technology,
Vietnam National University, Hanoi
144 Xuan Thuy, Cau Giay, Hanoi
trongld@vnu.edu.vn

ABSTRACT

Constructing and maintaining open source projects is not an easy work. Developers usually need to tackle a lot of bugs reported by users. The first step of the fixing progress is to find all relevant files respect to given reports. This step takes time and human resources. Motivating from this context, various learning-to-rank models were proposed in order to automatically rank files and generate suggestions for developers. The actual related files are expected to be appeared in high positions of the ranking. In the scope of this paper, a mean reciprocal rank optimization approach is investigated for the learning-to-rank relevant files for bug reports task. Given a bug report, the ranking of a source file is approximated by a function aggregating features which represent their relationship. The weights of these features are learned previously to maximize the mean reciprocal rank of known relevant files on training bug reports. In the experimental section, the introduced model is evaluated on three Java open source projects namely Tomcat, AspectJ and SWT. The three different versions of the model are also explored and compared to a recent state-of-the-art method in recommending related files for bug reports.

Keywords

Bug reports, Learning-to-Rank, Mean Reciprocal Rank

1. INTRODUCTION

Checking bug reports and fixing related files are usual tasks of developers in open-source projects. In the past, they need to search relevant files manually before solving the problem. It is no difficult if the bug contains class names or paths to files as the Figure 1 while it takes much more time for the textual bug like Figure 2, especially for bugs having more complex descriptions. The bigger the open-source project is, the larger number of bug reports developers need to deal. Thus, the requirement for automated methods to determine related files for bug reports becomes more and more seriously. These methods should give a small and suitable set of candidates for developers' references. Numerous learning-to-rank models in finding buggy files for bug reports are proposed such as [1, 3, 5, 9, 14, 15].

Because the bug report text and source code file are often informative, researches often focus on exploiting the textual features to link relevant files to bug reports. Kim et al. [3] simply proposed a two-phase recommendation model which relies on a binary Naive Bayes classifier. The classifier is learned using textual features of training bug reports and

Bug ID: 54703
Summary: Nullpointer exception in HttpParser.parseMediaType
Description: exception trace
java.lang.NullPointerException
at org.apache.tomcat.util.http.parser.HttpParser.parseMediaType(HttpParser.java:215)
at org.apache.tomcat.util.http.parser.MediaTypeCache.parse(MediaTypeCache.java:54)
at org.apache.catalina.connector.Response.setContentType(Response.java:806)
at org.apache.catalina.connector.Response.checkSpecialHeader(Response.java:1119)
at org.apache.catalina.connector.Response.setHeader(Response.java:1446)
at org.apache.catalina.connector.ResponseFacade.setHeader(ResponseFacade.java:535)

Figure 1: A bug report with class names

Bug ID: 55766
Summary: if path include relative char, eg: ../, classloader will not find the file
Description: I use ibatis in my project, config file path include '..', tomcat cannot find the file, command line show: Could not find resource ../ibatis/code/sendType.xml

Figure 2: A textual bug report

respective fixed source files. It is applied on new bugs and all source files to determine the relevances. The drawback of this approach is inapplicable for unknown source files. Taking advantages of topic model, Lukins et al. [5] introduced a hybrid model using LDA and Vector Space Model (VSM). Topic distributions of training bug reports and files are generated. Given a new bug, the ranked list of candidate is determined by sorting the similarity between the topic distribution of the bug report and a source file candidate. In 2011, Nguyen et al. [9] not only utilized the topic model but also a defect-proneness factor favors to frequently fixed files and files with large size. Exploring features from API documents and software repositories, Ye et al. [14] present a learning-to-rank model with 6 features. The significant enhancement show the effectiveness of the model. However, the authors utilized the SVM_{rank} toolkit which was witnessed several disadvantages mentioned in [2]. Additionally, it is clear that among relevant files of a bug report are usually contain explicit relationships. Starting from a file, we can travel to other relevant files via variable declarations. In other words, the recommender model does not require to suggest all correct files, just few most related ones.

Mean reciprocal rank (MRR), is first introduced in [13], is computed as the inverse of the rank of the first relevant item. As described in [8, 7], we can simultaneously learn parameters and maximize the rank of relevant items in the information retrieval context. Applying this great point on recommender systems, Shi et al. [10, 11] built a recom-

mender model via maximizing the smoothed mean reciprocal rank of users on relevant items. This approach just utilized relevant items (positive instances) in training and favor few (not all) relevant items as high position as possible. Hence, it inspires me to investigate a MRR-based optimization approach for learning to rank relevant file for bug reports as well as the presence of irrelevant files in promoting relevant files of bug reports. In the experimental section, the three versions of the MRR-based model are considered to explore the effect of related and unrelated files in favoring relevant files in the ranked list of a given bug report. The best model of the three models is compared to a recent state-of-the-art model LTR-SVM. All results are throughly discussed and explained later.

The rest of the paper is constructed as follows. Section 2 introduces a mean reciprocal rank maximization approach in finding buggy files for bug reports. The objective functions, features and the recommendation technique are described gradually to ensure a coherent order. This is followed by Section 3 which contains experimental settings as well as comparisons between models. A short discussion about related works is shown in Section 4 before findings and explanations are summarized in the last section.

2. A MEAN RECIPROCAL RANK MAXIMIZATION APPROACH

2.1 The smoothed Reciprocal Rank

Considering an open source project $\mathcal{P}$, let us denote B as the set of bug reports and $\mathcal{B}$ as the source files set $\mathcal{S}$. Given a bug report $b \in \mathcal{B}$, the reciprocal rank of a ranked list for this bug is inspired from [11]. It will favor a few relevant files at the very high position of the recommendation list.

$$RR_b = \sum_{i=1}^{N} \frac{Y_{bi}}{R_{bi}} \prod_{j=1}^{N} (1 - Y_{bj}\mathcal{I}(R_{bj} < R_{bi})) \tag{1}$$

where N is the total number of source files in $\mathcal{P}$; Y_{bi} is the binary relevance score of source file i to bug b; R_{bi} denotes the rank of source file i in the ranked list over all source files for the bug report b. The reciprocal rank and $\mathcal{I}(R_{bj} < R_{bi})$ are approximated using a logistic function:

$$\frac{1}{R_{bi}} \approx g(f_{bi}) \tag{2}$$

$$\mathcal{I}(R_{bj} < R_{bi}) \approx g(f_{bj} - f_{bi}) \tag{3}$$

where $g = 1/(1 + e^{-x})$; f_{bi} denotes the predictor function that aggregate features representing the relationship between a bug report b and a source file i to a predicted relevance score. It is defined as:

$$f_{bi} = \sum_{l=1}^{|W|} \phi_l(b, i) W_l \tag{4}$$

Each feature $\phi_l(b, i)$ measure a specific relationship between the source file i and the received bug report b while W_l is the respective feature parameter. The parameter W_l will be learned in order to maximize the mean reciprocal rank introduced later. Replacing Eq. (2) and Eq. (3) into Eq. (1), the smooth version of RR_b will be:

$$RR_b \approx \sum_{i=1}^{N} Y_{bi} g(f_{bi}) \prod_{j=1}^{N} (1 - Y_{bj} g(f_{bj} - f_{bi})) \tag{5}$$

Let us denote the number of relevant files for bug report b in the given data collection is n_b^+. So RR_b/n_b^+ is the mean reciprocal rank of bug report b on all source files. Apply the Jensen's inequality and the concavity of the logarithm function, we obtain the lower bound of the natural logarithm of the mean reciprocal rank:

$$\ln(\frac{1}{n_b^+} RR_b) \geqslant \frac{1}{n_b^+} \sum_{i=1}^{N} Y_{bi} (\ln g(f_{bi}) + \sum_{j=1}^{N} \ln(1 - Y_{bj} g(f_{bj} - f_{bi}))) \tag{6}$$

Neglecting the constraint $1/n_b^+$, the objective function for the mean reciprocal rank can be expressed as follow:

$$L(b, S) = \sum_{i=1}^{N} Y_{bi} (\ln g(f_{bi}) + \sum_{j=1}^{N} ln(1 - Y_{bj} g(f_{bj} - f_{bi}))) \tag{7}$$

Adding irrelevant files

In [11], Shi et al. actually use the relevant items set in their objective function as Eq. (1). With the objective of investigating the effect of irrelevant files to the learning-to-rank model, I consider the presence of unrelated files in the reciprocal rank equation. Extending from Eq. (1), we have the new reciprocal rank for a given bug report b

$$RR_b = \sum_{i=1}^{N} [\frac{Y_{bi}}{R_{bi}} \prod_{j=1}^{N} (1 - Y_{bj}\mathcal{I}(R_{bj} < R_{bi})) \times \prod_{k=1}^{N} (1 - \overline{Y}_{bk}\mathcal{I}(R_{bi} < R_{bk}))] \tag{8}$$

where $\overline{Y}_{bk} = 1$ if the source file k and the bug report b are irrelevant, otherwise 0. The primary idea is to favor more relevant files which are ranked higher than a number of irrelevant ones. Following the same steps, we obtain the objective function for the mean reciprocal rank with irrelevant files as:

$$L(b, S) = \sum_{i=1}^{N} Y_{bi} [\ln g(f_{bi}) + \sum_{j=1}^{N} \ln(1 - Y_{bj} g(f_{bj} - f_{bi})) + \sum_{k=1}^{N} \ln(1 - \overline{Y}_{bk} g(f_{bi} - f_{bk}))] \tag{9}$$

Then the objective function for all bug reports in the project $\mathcal{P}$ is expressed as:

$$L(B, S) = \sum_{b \in B} \sum_{i=1}^{N} Y_{bi} [\ln g(f_{bi}) + \sum_{j=1}^{N} \ln(1 - Y_{bj} g(f_{bj} - f_{bi}) + \sum_{k=1}^{N} \ln(1 - \overline{Y}_{bk} g(f_{bi} - f_{bk}))] \tag{10}$$

To avoid over-fitting problem as well as controlling the complexity of the learning-to-rank model, the regularization

term should be introduced. We have the final objective function as:

$$F(B,S) = \sum_{b\in B}\sum_{i=1}^{N} Y_{bi}[\ln g(f_{bi}) + \sum_{j=1}^{N}\ln(1 - Y_{bj}g(f_{bj} - f_{bi}) + \sum_{k=1}^{N}\ln(1 - \overline{Y}_{bk}g(f_{bi} - f_{bk}))] - \frac{\lambda}{2}\|W\|^2 \quad (11)$$

in which λ is the regularization coefficient and $\|W\|$ denotes the Frobenius norm of W. Using this objective function we can learn the parameter W in order to maximize the mean reciprocal rank over all bug reports in the project $\mathcal{P}$

2.2 Features

In the scope of this work, I only want to inspect the efficiency of the mean reciprocal rank-based approach in finding relevant files for bug reports so that I utilize the same six features proposed in [14]. Given a bug report b and a source file s, these features are described as:

Surface Lexical Similarity

This feature represent the textual similarity between the bug report b and the source file s. In the worst case, the source file is large while just a segment of code is related to the bug report, the authors compute the similarity for each method m in s with the bug report b.

$$\phi_1(b,s) = \max(\{sim(b,s)\} \cup \{sim(b,m)|m \in s\}) \quad (12)$$

where $sim(b,s), sim(b,m)$ are token cosine similarity.

API-Enriched Lexical Similarity

There is a case that the bug report and the relevant buggy file have very few common tokens. This problem triggers in unmeaning 0 cosine similarity for clarifying relatedness. Fortunately, the authors found that there are numerous of cases witnessing the relevance at abstract levels of the source file. They extract the API description of super classes and interfaces (if available) of the source file and compute the token cosine similarity to the bug report.

$$\phi_2(b,s) = \max(\{sim(b,s.api)\} \cup \{sim(b,m.api)|m \in s\}) \quad (13)$$

where $m.api$ is the API description of all super classes and interfaces which are declared in the method m; $s.api = \cup_{m\in s} m.api$

Collaborative Filtering Score

The denotation $br(b,s)$ is the set of bug reports for which file s was fixed before b was reported. This feature measures the similarity of the considering bug report to the previous ones which relate to the considering source file.

$$\phi_3(b,s) = sim(b, br(b,s)) \quad (14)$$

Class Name Similarity

The easiest way to observe the relevance is to check the presence of the class name of the source file in the bug report. The longer the class name is, the stronger the relationship between the source file and the bug report is.

$$\phi_4(b,s) = \begin{cases} |s.class| & \text{if } s.class \in \text{b} \\ 0 & \text{otherwise} \end{cases} \quad (15)$$

Bug-Fixing Recency

The authors claims that the recent changed source file have higher probability containing bugs than a file was last fixed long time in the past or never fixed. Denoting $last(b,s) \in br(b,s)$ be the most recently fixed bug and $b.month$ as the month when the bug was reported. The feature is defined as:

$$\phi_5(b,s) = (b.month - last(b,s).month + 1)^{-1} \quad (16)$$

Bug-Fixing Frequency

The fixing frequency of a given source file is informative. It might imply the complexity of the source file.

$$\phi_6(b,s) = |br(b,s)| \quad (17)$$

Finally, all features are scaled in order to make them becoming more comparable with each other. Considering an arbitrary feature ϕ, $\phi.min$ and $\phi.max$ are the minimum and maximum computed values in the training dataset. In order to maintain the consistency in both training and testing datasets, the value of the feature ϕ is re-computed as:

$$\phi' = \begin{cases} 0 & \text{if } \phi < \phi.min \\ \frac{\phi - \phi.min}{\phi.max - \phi.min} & \text{if } \phi.min \le \phi \le \phi.max \\ 1 & \text{if } \phi > \phi.min \end{cases} \quad (18)$$

2.3 Optimization

I apply the gradient ascent method to maximize the objective function in Eq. (11). Given a bug report b and a source file i, the gradient for each parameter W_l respecting to the feature l is computed as:

$$\frac{\partial F}{\partial W_l} = \sum_{i=1}^{N} Y_{bi}[\ln g(-f_{bi})\phi_l(b,i) + \sum_{j=1}^{N}\frac{Y_{bj}g'(f_{bj} - f_{bi})}{\ln(1 - Y_{bj}g(f_{bj} - f_{bi}))}(\phi_l(b,i) - \phi_l(b,j)) + \sum_{k=1}^{N}\frac{\overline{Y}_{bk}g'(f_{bi} - f_{bk})}{\ln(1 - \overline{Y}_{bk}g(f_{bi} - f_{bk}))}(\phi_l(b,k) - \phi_l(b,i))] - \lambda W_l \quad (19)$$

where $g'(x)$, the derivative of $g(x)$, could be calculated as $g'(x) = g(x)(1 - g(x))$ while $g(-x) = g'(x)/g(x)$. The learning process is described in the **Algorithm 1**, γ is the learning rate.

2.4 Relevant Files Recommendation

According to the **Algorithm 1**, the parameter W is learned with the objective of maximizing the mean reciprocal rank. For a given bug b and a source file i, the relevance score f_{bi} could be computed via the Eq. (4). Likewise, the reciprocal rank R_{bi} is approximated via the Eq. (5). We have the rank of the source file i in the ranked list for the bug report b:

$$R_{bi} \approx \frac{1}{g(f_{bi})} \quad (20)$$

Algorithm 1: Learning Algorithm for LTR-MRR

Input : Training bugs set B and source files set S
Output: The learned parameter W
foreach *(b, s)* **do**
  % Compute $\phi_l(b, s)$ repsects to Eq. (12), (13), (14), (15), (16), (17) ;
  $b \in B, s \in S, l = 1, 2, ..., |W|$
end
Initialize $W^{(0)}$ randomly;
repeat
  for *l = 1, 2, ...,* $|W|$ **do**
    % Update W_f;
    $W_l^{(t+1)} = W_l^{(t)} + \gamma \frac{\partial F}{\partial W_l}$ based on Eq. (19) ;
  end
  t = t + 1;
until $t \geqslant$ *itermax or converged*;

This rank is utilized to suggest the top K files as relevant candidate.

3. EXPERIMENTS

As mentioned in the Section 1 as well as the Sub-Section 2.2, this work is to explore the efficiency of the mean reciprocal rank-based approach in recommending relevant files to bug reports. In this section, I present several comparisons between three different version of the proposed model as well as a recent stat-of-the-art learning-to-rank relevant files fore bug reports method [14]. Based on results, I give a discussion to clarify my empirical study about the learning-to-rank related files for bugs task.

3.1 Datasets & Methodology

Datasets

There are numerous of open-source projects could be utilized in the experimental section. On the ground of the availability and efficiency of projects, I select three Java projects for evaluating the introduced model in the sub section 2.1, namely

- AspectJ [1]: an aspect-oriented programming extension for Java
- SWT [2]: a widget toolkit for Java
- Tomcat [3]: a web application server and servlet container.

Ye et al. [14] collected bug reports as well as respective relevant files for the three datasets. Subsequently, they share these files and their results publicly [4]. The *AspectJ*, *SWT*, *Tomcat* contains 593, 4151, 1056 bug reports respectively. Thank to the authors' instruction, the features listed in the subsection 2.2 are analyzed properly for the three datasets.

Methodology

For all datasets except *AspectJ*, the bug reports are sorted by their report timestamp descent order before separating into 10 equally size folds ($fold_1, fold_2, ..., fold_{10}$). Due to the small number of bug reports, the bugs of *AspectJ* are just ordered and divided into 3 equally size folds. In the later experiments, the learning-to-rank model will be trained on $fold_{k+1}$ and tested on $fold_k$ for all $k \leq 9$ ($k \leq 2$ for *AspectJ*). The coefficients λ and γ using in the model are tunned on each training fold with 4 different values $\{0.01, 0.001, 0.001, 0.0001\}$. The pair of (λ, γ) generates the highest mean reciprocal rank in Eq. (11) on training data will be utilized in the testing phase then. Equally important, because the number of irrelevant files could be very large, it could affect on the computational cost of the introduced model. Therefore, for each bug report b, I only sample the top $M \in \{50, 100, 200, 300\}$) irrelevant files which are most similar to the bug report b in term of the surface lexical feature. Finally, the overall performance of the model will be the average of results over all testing folds using the following evaluation metrics:

- *Accuracy@k* measures how many percentages of testing bug reports having at least one correct recommendation in the top k ranked files.
- *Mean Average Precision (MAP)* [6] represents the mean of Average Precision ($AvgP$) values obtained on all testing bug reports set $\mathcal{B}_{test}$

$$MAP = \sum_{b \in \mathcal{B}_{test}} \frac{AvgP(b)}{|\mathcal{B}_{test}|}, AvgP = \sum_{k \in K} \frac{Prec@k}{K} \tag{21}$$

$$Prec@k = \frac{\#\text{ of relevant files in top k}}{k} \tag{22}$$

- *Mean Reciprocal Rank (MRR)* [13] is relied on the inverse of the rank of the first relevant file $first_b$ for a given bug report b. It is defined as:

$$MRR = \frac{1}{|\mathcal{B}_{test}|} \sum_{b \in \mathcal{B}_{test}} \frac{1}{first_b} \tag{23}$$

Models In the later subsection, I compare the performance on the four following models:

- LTR-MRR-P: The mean reciprocal rank-based maximization approach using only the training relevant files in the objective function as Eq. (7)
- LTR-MRR-N: The mean reciprocal rank-based maximization approach without the training relevant files in the objective function based on Eq. (11)
- LTR-MRR: The mean reciprocal rank-based maximization approach using both the training relevant and irrelevant files in the objective function as Eq. (11)
- LTR-SVM: The state-of-the-art learning-to-rank model introduced in [14]. The ranking result is mentioned in the section 3.1

[1]https://eclipse.org/aspectj
[2]https://eclipse.org/swt
[3]http://tomcat.apache.org
[4]http://files.figshare.com/1656551/dataset.zip

Table 1: Accuracy@k comparison on Tomcat using various top K irrelevant files

k	LTR-MRR-P	LTR-MRR				LTR-MRR-N			
		50	100	200	300	50	100	200	300
1	31.41	**34.29**	33.12	33.01	34.94	16.77	16.77	13.68	13.68
5	56.73	**58.33**	57.26	57.80	56.73	36.11	36.10	18.48	18.48
10	67.41	**67.52**	66.77	66.45	66.67	45.08	45.09	21.05	21.05
20	76.82	**78.53**	78.42	77.88	75.11	60.15	60.13	23.81	23.82

Table 2: MRR and MAP comparisons on Tomcat using various top K irrelevant files

	LTR-MRR-P	LTR-MRR				LTR-MRR-N			
		50	100	200	300	50	100	200	300
MRR	0.435	**0.458**	0.448	0.561	0.460	0.270	0.270	0.160	0.160
MAP	0.107	**0.111**	0.109	0.109	0.108	0.065	0.065	0.042	0.042

The first three models are to investigate the impact of relevant and irrelevant files in favoring the target relevant files for a given bug report. Subsequently, the best one will be compared to the LTR-SVM model in order to clarify the efficiency of the learning-to-rank model based on the mean reciprocal rank.

3.2 Comparisons & Discussion

The recommendation performance of the three models LTR-MRR-P, LTR-MRR-N, LTR-MRR on the *Tomcat* dataset are shown in the Table 1 and Table 2. Various numbers of the top K $\in \{50, 100, 200, 300\}$ irrelevant files are tested for the two models LTR-MRR-N, LTR-MRR .There are two main observations from the results including: Firstly, the LTR-MRR model using the top 50 is witnessed the best performance in both *Accuracy@k*, MAP and MRR. Excepting the top 10, it outperforms the others significantly with $p \leq 0.05$. It is not bad when over 70% suggestions at top 20 are correct. This phenomena shows the efficiency to promote relevant files using both (other) relevant and irrelevant files. Secondly, the LTR-MRR-N model seems to be worst. There is no much difference when use various number of the top unrelated files. The problem could be caused by the similar coverage of the unrelated files having negative effect on the related buggy files. Likewise, I conduct the same experiments on *AspectJ* and *SWT* dataset. The LTR-MRR model still shows it stable trend comparing the the two remaining models. With these analysis, I decide to use the LTR-MRR model using top 50 irrelevant files as the representative for further comparisons.

The Table 3 and Table 4 illustrate the comparison between the LTR-SVM model and the LTR-MRR model in terms of *Accuracy@k*, MRR and MAP. Unfortunately, the LTR-SVM model outnumbers the LTR-MRR number excepts the top 20 in the *ApsectJ* experiment. The gaps are significant and consistent over datasets and measurements. This issue reckons that the MRR-based model (LTR-MRR) does not rank files as well as the SVM-based model (LTR-SVM). In my opinion, the main reason is the difference in using irrelevant files. The LTR-SVM model concurrently promote the relevant files and penalize the irrelevant ones via the pairing comparison $f(b, s_{relevant} > f(b, s_{irrelevant}))$ while the other just utilize the irrelevance to favor the irrelevant files only. Additionally, in the Table 5 show the average learned parameters on the three datasets. It is clear that there are few features having more important roles than others, especially the surface lexical similarity w_1. In other words, the rank of a source file for a given bug is just dependent on several main features. This problem might not happen in the inspired paper [11] where latent factors were utilized. With mentioned points, the LTR-MRR model could not differentiate the relevant and irrelevant file in the ranking properly.

4. RELATED WORKS

Here, I survey the literature in learning to rank relevant files for bug reports and MRR-based optimization approaches

Learning to rank relevant files for bug reports It is a popular topic in software mining research. The accuracy performance is improved by time, become better and better. In 2011, Nguyen et al. [9] apply topic modeling technique to determine buggy files for given bug reports. The primary idea is to learn the topic distribution on training bug reports and source files. For a new bug report, its topic distribution is generated using the trained LDA model before comparing the topic distributions of all source file candidates. However, this model only works on fixed version projects which have no changes in source files. With the objective of dealing the task as a classification problem, Kim et al. [3] built a binary Naive Bayes classifier using textual features on both training bug reports and their respective files. For a new bug report, this classifier is utilized to determine the relevance of this bug with with all source file candidates. Undoubtedly, unknown or never-fixed source files could not be a recommendation. In [14], Ye et al. propose a learning-to-rank models using domain knowledge. They introduced 6 features to estimate parameters using SVM^{rank} toolkit [5] where pairing comparisons are exploited. Subsequently, these parameters are utilized to distinguish the relevant and irrelevant files for a given bug report. In 2016, Ye et al. publish the extended version [15] with 17 features. Regardless of significant improvements, the drawback of using SVM^{rank} is discussed in [2]. With the objective of exploiting latent relationship between bug reports and source files, An et al. [4] proposed a hybrid model taking advantages of deep neural network (DNN) and an information retrieval technique rVSM. They utilized three DNN models in order to bridge the lexical gap in which features of different types in bug report and source code are tackled in diverse spaces; combine features in lower layer and perform dimension reduction for feature vectors.

[5]http://cs.cornell.edu/people/tj/svm_light/svm_rank.html

Table 3: Accuracy@k comparison between LTR-SVM and LTR-MRR (using top 50 irrelevant files)

k	Tomcat		AspectJ		SWT	
	LTR-SVM	LTR-MRR	LTR-SVM	LTR-MRR	LTR-SVM	LTR-MRR
1	45.94	34.29	27.88	21.23	31.13	14.96
5	67.63	58.33	56.78	46.28	61.77	36.43
10	77.14	67.52	68.54	63.68	73.73	50.83
20	83.87	76.82	75.19	75.45	83.19	65.44

Table 4: MRR and MAP comparisons between LTR-SVM and LTR-MRR (using top 50 irrelevant files)

	Tomcat		AspectJ		SWT	
	LTR-SVM	LTR-MRR	LTR-SVM	LTR-MRR	LTR-SVM	LTR-MRR
MRR	0.561	0.435	0.411	0.355	0.450	0.242
MAP	0.130	0.107	0.108	0.100	0.118	0.071

Table 5: The average model parameters

Project	w_1	w_2	w_3	w_4	w_5	w_6
AspectJ	1.61	0.23	0.40	0.18	0.21	0.19
SWT	18.54	7.83	7.47	4.08	3.93	3.81
Tomcat	3.41	0.72	1.07	1.14	0.39	0.68

Deep learning is a strong technique but it is inefficient to update models frequently. Recently, Le et al. [1] presented a learning-to-rank based fault localization method using likely invariants and suspiciousness as features. This model uses a subset of test cases including both falling and passing tests. Through logging all execution traces, the likely invariants and suspiciousness features scores are determined and utilized to rank suspicious methods in a given fault localization context. In [12], Tian et al. considered the bug report assignee recommendation task. They extracted 16 activity-based and location-based features before using the rankSVM package [6] to learn parameters.

MRR-based maximization approaches In 2005, Metzler et al. [8] proposed a direct maximization of ranked-based metrics for information retrieval. Regardless of imposing an ordinal response ("relevant" and "non-relevant"), they focus on finding parameters to maximize ranking-based evaluation metrics. This method is benefit for information retrieval, collaborative filtering as well as general classification tasks. Likewise, Mcfee et al. [7] presented a general metric learning algorithm based on the structural SVM framework. This work bridge the gap between metric learning and ranking. It supports various ranking measures including Precision-at-k, Average Precision (AP) and Mean Reciprocal Rank (MRR). Inspiring from these works, Shi et al. [11] built a recommender model, named *CLiMF*, via maximizing a smoothed mean reciprocal rank of users on relevant items. Given an user, the rank of an item to this user and comparisons between items' ranks are approximated using the sigmoid function and latent vectors. The model has close relationships with a lot of state-of-the-art recommendation models. In 2013, *xCLiMF*[10], an extended version of *CLiMF*, was introduced with the objective of building a recommendation models by optimizing Expected Reciprocal Rank to consider user feedback with multiple level of relevance. The advantages of the *CLiMF* and *xCLiMF* namely: do not require irrelevant items (negative samples) and directly estimate the rank of items in ranked list for a given user.

[6]https://www.csie.ntu.edu.tw/ cjlin/papers/ranksvm

5. CONCLUSIONS

In the scope of this work, I investigated a learning-to-rank model in finding relevant files for bug reports based on the mean reciprocal rank. The main idea is to directly enhance the position of few relevant files in the ranked list of respective bug reports. We could travel to the remaining files using variable declarations. Regardless of witnessing improvements when incorporating irrelevant files in training process, the overall performance of the introduced model on three datasets cannot outperform the recent state-of-the-art method [14] using SVM^{rank}. This problem can be explained by the two reasons namely: i) The model just favor the relevant files while it does not penalize the irrelevant ones in training process; ii) The limited number of features as well as their unfair roles in datasets. Hence, the model cannot distinguish the differences between related and unrelated files in order to generate good recommendations.

6. REFERENCES

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

[2] P. Donmez and J. G. Carbonell. Optimizing estimated loss reduction for active sampling in rank learning. In *Proceedings of the 25th international conference on Machine learning*, pages 248–255. ACM, 2008.

[3] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.

[4] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.

[5] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *2008 15th Working Conference on Reverse Engineering*, pages 155–164. IEEE, 2008.

[6] C. D. Manning, P. Raghavan, and H. Schütze. Introduction to information retrieval. 2008.

[7] B. McFee and G. R. Lanckriet. Metric learning to rank. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 775–782, 2010.

[8] D. A. Metzler, W. B. Croft, and A. McCallum. Direct maximization of rank-based metrics for information retrieval. 2005.

[9] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 263–272. IEEE, 2011.

[10] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, and A. Hanjalic. xclimf: optimizing expected reciprocal rank for data with multiple levels of relevance. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 431–434. ACM, 2013.

[11] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, N. Oliver, and A. Hanjalic. Climf: learning to maximize reciprocal rank with collaborative less-is-more filtering. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 139–146. ACM, 2012.

[12] Y. Tian, D. Wijedasa, D. Lo, and C. L. Gouesy. Learning to rank for bug report assignee recommendation. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May 2016.

[13] E. M. Voorhees. The trec question answering track. *Natural Language Engineering*, 7(04):361–378, 2001.

[14] X. Ye, R. Bunescu, and C. Liu. 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*, pages 689–699. ACM, 2014.

[15] X. Ye, R. Bunescu, and C. Liu. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering*, 42(4):379–402, 2016.