Learning to Rank for Question-Oriented Software Text Retrieval

Yanzhen Zou*[1,2,3], Ting Ye*[1,2], Yangyang Lu[1,2], John Mylopoulos[3], Lu Zhang[1,2]
[1]Software Institute, School of Electronics Engineering and Computer Science, Peking University, China
[2]Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
[3]Department of Computer Science, University of Toronto, ON, Canada
{zouyz, yeting, luyy, zhanglucs}@pku.edu.cn; jm@cs.toronto.edu

Abstract—**Question-oriented text retrieval, aka natural language-based text retrieval, has been widely used in software engineering. Earlier work has concluded that questions with the same keywords but different interrogatives (such as how, what) should result in different answers. But what is the difference? How to identify the right answers to a question? In this paper, we propose to investigate the "answer style" of software questions with different interrogatives. Towards this end, we build classifiers in a software text repository and propose a re-ranking approach to refine search results. The classifiers are trained by over 16,000 answers from the *StackOverflow* forum. Each answer is labeled accurately by its question's explicit or implicit interrogatives. We have evaluated the performance of our classifiers and the refinement of our re-ranking approach in software text retrieval. Our approach results in 13.1% and 12.6% respectively improvement with respect to text retrieval criteria nDCG@1 and nDCG@10 compared to the baseline. We also apply our approach to FAQs of 7 open source projects and show 13.2% improvement with respect to nDCG@1. The results of our experiments suggest that our approach could find answers to FAQs more precisely.**

I. Introduction

Text retrieval (TR) is often used to help developers search for software artifacts in many software engineering tasks. It has been successfully applied to address more than twenty tasks , including software change requests, concept/feature/concern location, impact analysis, code search and reuse, traceability link recovery, etc [4], [22], [26], [39], [40].

In existing software text retrieval systems (or search engines), a lot of work has been done on keyword matching [3], [9], [10]. Usually, given a query, the search engine returns the relevant documents that contain keywords in the query. However, there are so many search results containing the same keywords, resulting in poor results for large software text repositories. For example, more than 820,000 question-answer pairs are relevant to the keyword "index" in the *StackOverflow* forum.

Actually, questions differ not only with respect to keywords, but also interrogatives. Different interrogatives should result in different answers in software text retrieval. For example, a "how to" calls for instructions[29], "what" asks for the meaning of a particular code entity[18], and "where" asks for the location of an entity.

In this paper, we investigate how to leverage these interrogatives in question-oriented software text retrieval. This is an extended version of our previous new idea paper [36]. Our basic hypothesis is that, there must be an answer style for each interrogative. So we build a software document classification system to select discriminative features for answers of "how-to" questions or other interrogative questions. The classification system is trained by more than 16,000 answers from *StackOverflow*, the largest and programming-specific forum. When representing these answer documents as feature vectors, we consider both text features, such as words, and non-text features, such as code, link and length. Then we apply our classifiers to re-rank the search results in a large software text retrieval system. The classifiers are used to identify whether a document has the answer style of the input question. Documents having the required features are ranked higher. As a result, we substantially improve the search results beyond the baseline of keyword-based retrieval.

*These authors contributed equally to this paper.

Compared to the state-of-the-art, our work makes the following contributions:

1) We build six interrogative-guided software document classifiers for "howto", "what", "where", "which", "whether" and "why" questions' answers respectively. Using them, we can tell whether a document has the answer style of the interrogative questions. As a result, all documents in the repository are automatically classified and labelled using interrogatives.

2) We integrate our classifiers with a well-known open source text retrieval engine, *Lucene*, and propose a new re-ranking approach that re-ranks each search result according to its classification score. If a software text is classified consistently with a question, it gains a higher score, and conversely.

3) We build our classifiers with data derived from *StackOverflow*, but have also established that our classifiers are project independent and text source independent. To do this, we applied our approach to 7 well-known open source projects to show that our search engine could find their FAQs answers more precisely in software text repositories.

The rest of the paper is organized as follows. Section 2 describes a motivating example. Section 3 gives an overview of our approach. Section 4 and Section 5 explain the details of our classifier building process and re-ranking approach respectively. Section 6 presents experiments that evaluate our approach. Related work is presented in Section 7. We conclude and discuss future work in Section 8.

II. Motivating Example

In this section, we present two sample questions selected from *StackOverflow* to further motivate our research. These two questions have similar keywords but different interrogatives. The first question is “Where is StandardAnalyzer?”[1], while the second question is “How do I use StandardAnalyzer with TermQuery?”[2]. Obviously, the questioner wants to know the position where StandardAnalyzer is located in question 1 but how to use StandardAnalyzer in question 2. We apply these questions to a keyword based search engine, *Lucene*, and find that: 1) these two questions have 22 retrieval results in common in top-30 search results. 2) the answer of question 2 (A2) ranked higher than the best answer we want (A1) when searching the answer of question 1. These results indicate that more precise retrieval techniques are needed.

TABLE I: Two Sample Questions Selected from StackOverflow

Q1: Lucene 3: Where is StandardAnalyzer?
A1: It looks like they are moving the StandardAnalyzer (and the related stuff) under org.apache.lucene.modules.analysis.* I don’t know the reason though. You can find the StandardAnalyzer in the SVN Trunk here: <a href= http://svn.apache.org/...> ... </a>
Q2: How do I use StandardAnalyzer with TermQuery?
A2: The established way to get the token text is with <code> token.termText() </code> - that API’s been there forever. And yes, you’ll need to specify a field name to both the <code> Analyzer </code> and the <code> Term </code>; I think that’s considered normal.

Actually, questions with the same keywords but different interrogatives should be answered differently. In table I, we present the best answers of the two questions from *StackOverflow*. It can been seen that: in the answer of the “where” question (Q1), there is a link to access the class’s location. And in the answer of the “how” question (Q2), there are more code snippets and the description contains some class or method name as well.

Then, how to find these differences? In other words, what is the answer style of “where” questions versus that of “how” questions? Fortunately, there are many software questions and answers on the Internet where we can learn the “answer style” for each interrogative. And we pick our learning examples (question-answer pairs) from *StackOverflow*. We label each question-answer pair according to its interrogative. The feature selection method can then tell us what the discriminative features are in the documents when answering each type of question. The discriminative features are used for training the classifiers. Then we apply our classifiers to software text retrieval and propose a re-rank approach to refine search results.

III. Approach Overview

An overview of our approach is shown in figure 1, it is composed of the following two parts: off-line classifier building and on-line search results re-ranking.

[1]http://stackoverflow.com/questions/3080888/lucene-3-where-is-standardanalyzer

[2]http://stackoverflow.com/questions/1390088/how-do-i-use-standardanalyzer-with-termquery

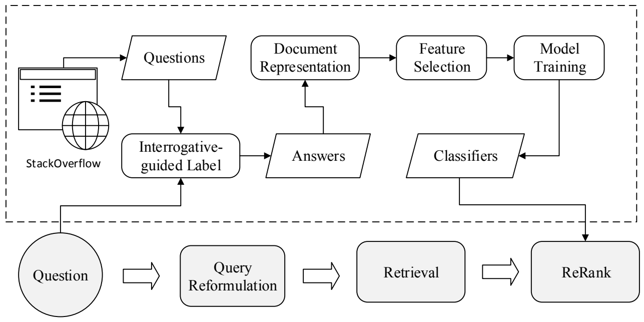

Fig. 1: Approach Overview

Off-line Classifier Building: It aims to find the answer style for each interrogative and build corresponding document classifier. First, all questions collected are labeled based on their interrogatives (e.g., how, why, where etc.). Cross validation is performed here to ensure the accuracy of the labeled data. Naturally, their answers are labeled with the same tags. Second, a document representation component represents each answer document as a feature vector. And these features are weighted by different feature weight methods. Third, a feature selection component is used to learn the most important features, which also aims to avoid the over-fitting in classifier training process. Finally, the discriminative features are used to build document classifiers.

On-line Result Re-Ranking: This part takes advantage of the built classifiers to re-rank the search results in software text retrieval. In application, some candidate answer documents would be returned by a text retrieval engine when a user inputs a question. Then, we use the classifier to re-rank these search results based on the following rules: if a software text is classified as the same type as the question, it gains higher score, and conversely. As a result, those documents with the required features are re-ranked towards the top, while those do not have the required features are re-ranked lower down.

In the following two sections, we introduce our approach step by step.

IV. Classifier Building

In this section, we describe the process of Classifier Building. This work is composed of question labeling, document representation, feature selection and model training.

A. Question Labeling

According to our observations, the questions usually have 8 kinds of interrogatives, including “how”, “what”, “why”, etc. We propose to label questions according to their interrogatives. In table II, we illustrate each tag and how a question can be labeled with this tag.

Based on these interrogative tags, we collected question and answer pairs from *StackOverflow* to set up learning examples. Here a question may be labeled with one or more interrogative tags. The detailed dataset setting-up process is described in section VI-A. To ensure that every question is labeled with the right tag, we asked 3 Phd students, 4 master candidates and 3 undergraduate students to do the question

TABLE II: Question Tags

Tag	Description
howto	Asks for instructions, which usually begins with word "how". e.g., *How to update a Lucene.NET index?*
what	Asks what a variable is. e.g. *What is the segment in Lucene?*
which	Asks for an advice when meeting multiple choice. e.g. *Which is the best choice to indexing a Boolean value in lucene?*
where	Asks for location(path, url, directory etc). e.g. *Where is StandardAnalyzer?*
why	Asks for reasons. e.g.*Why Lucene merge indexes?*
when	Asks about the time at which things happen. e.g. *When should one stop using Lucene and graduate to Solr?*
who	Asks about the name or identity of a person or group of people. e.g. *Who created the C programming language?*
whether	Asks for yes or no. e.g. *Should an index be optimized after incremental indexes in Lucene?*
others	The rest questions which are hard to be labeled.

labeling work manually. One third of the questions are cross validated so as to minimize mistakes in this process.

The labeled dataset is shown in table IV. The most frequent interrogative type is "howto", followed by "whether", "what", "why", "which" and "where". Few "when" (0.23%) and "who" (0.091%) type questions are collected. This result is general, modulo some variations, in software engineering question-and-answer forums. For example, in 2009, Harper et al. [11] indicated that, among the questions of Yahoo! Answers, the rate of "who" questions was 13.9% and the rate of "when" questions was 17.0%. As the amount of "when" and "who" is much less than the amount of other types, we do not build classifiers for these types and focus on the 6 main types, i.e., "howto", "whether", "what", "why", "which", "where".

Please note that a question post in *StackOverflow* may contain some description sentences besides a question title. These description sentences may include another question which has new interrogatives. That is to say, some question may have more than one tags. For example, the question *"I want to know what is the VInt in Lucene ? I read this article , but i don't understand what is it and where does Lucene use it ? Why Lucene doesn't use simple integer or big integer ? Thanks ."*[3] would be labeled as "what", "where" and "why".

B. Document Representation

Once a question is labeled, its answers are labeled with the same interrogative tags. Then we use the answer set to learn the "answer style" for each interrogative. For this purpose, we need to parse every answer document into a feature vector.

1) Feature Extraction: We use two kinds of features, text features and non-text features in our document representation.

A)Text Features: The text of an answer gives an explanation to the question, thus some words may closely relate to the question type. The text features are automatically extracted from document information after stop-word removal and stemming [21]. The stop words include three parts: 1) Common stop words, such as, "is", "are", "am", "would", etc. 2) Java reserved words, such as, "public", "static", "void", etc. 3) Labels defined by *StackOverflow*, such as "<code>", "<img>", etc. Stemming reduces a word to its root. In our work, we choose the snowball stemming algorithm[4] [24] to reduce the word to its root. Note that the root of noun and the root of verb are different. For example, "explains" and "explained" reduce to the stem "explain", but "explanation" and "explanations" reduce to the stem "explan".

B)Non-text Features: The answer document usually contains code snippets and links. Empirically, an answer of a *howto* question is more likely to contain code snippets to explain the process step by step. Therefore, we investigate the XML tags used by *StackOverflow*, where <code> and </code> pairs indicate a code snippets, <a> and </a> include a link. By this way, we extract some non-text features from each answer document.

TABLE III: Features for Document Representation

ID	Feature Description
W_1 *to* W_n	Each feature represents a stemmed word in the answer set. Each feature has a value corresponding to the number of times the word appears in the answer.
Cod-num	n. of code snippets
Cod-avc	Average code length
Cod-mac	Maximum code length
Cod-mic	Minimum code length
Cod-sdc	Code length standard deviation
Cod-malc	Maximum code lines
Cod-milc	Minimum code lines
Cod-sdlc	Code lines standard deviation
Cod-clc	n. of code lines
Link-xlc	n. of links to external sources
Link-ilc	n. of links to other query/answer in the forum
Link-lc	n. of links
Len-wc	n. of words
Len-sc	n. of sentences
Len-cc	n. of characters
Len-pc	n. of paragraphs

Table III shows the features we used in the document representation. Here W_1 to W_n are text features, each represents a stemmed word in the answer set; the names begin with "Cod" are code related features; the names begin with "Link" are link related features. We also add some statistical features to investigate the effects of document length. As a result, every answer document can be represented as $\overrightarrow{D_j} = \{d_{1j}, d_{2j}, ...d_{nj}\}$, where d_{ij} refers to either a textual or a non-textual feature.

2) Feature Weighting: There are various alternatives and enhancements in constructing the $\overrightarrow{D_j}$ vector. In our work, we apply two feature weighting methods: term frequency [6] and term frequency-inverse document frequency [21].

Term frequency (TF): The *ith* component of the feature vector is the number of times that term t_i appears in the answer document. Let $\mathrm{v}(t_i, \overrightarrow{D_j})$ denote the frequency of term t_i in document D_j. The feature vector is normalized to unit length

[3]http://stackoverflow.com/questions/2752612/what-is-the-vint-in-lucene

[4]http://snowball.tartarus.org/

after calculating each x_{ij} value.

$$x_{ij} = \text{tf}(t_i, \overrightarrow{D_j}) = \frac{\text{v}(t_i, \overrightarrow{D_j})}{\max\{\text{v}(t_k, \overrightarrow{D_j}) : t_k \in \overrightarrow{D_j}\}} \quad (1)$$

Term frequency - inverse document frequency (TF-IDF): Instead of using the term frequency value for each answer document, this method also considers how frequently a term appears in the document set. The inverse document frequency is a measure of how much information one feature provides, that is, whether the term is common or rare across all documents. Let N be the total number of documents in the corpus. Also, let $|\{\overrightarrow{D_j} \in D : t_i \in \overrightarrow{D_j}\}|$ be the number of documents where the feature t_i appears (i.e., $\text{tf}(t_i, \overrightarrow{D_j}) \neq 0$). We adjust the denominator to $1 + |\{\overrightarrow{D_j} \in D : t_i \in \overrightarrow{D_j}\}|$ to smooth the feature value and avoid a division-by-zero. The feature vector is also normalized to unit length. The idf method and tf-idf method are calculated as follows:

$$\text{idf}(t_i, D) = \log \frac{N}{1 + |\{\overrightarrow{D_j} \in D : t_i \in \overrightarrow{D_j}\}|} \quad (2)$$

$$x_{ij} = \text{tfidf}(t_i, \overrightarrow{D_j}, D) = \text{tf}(t_i, \overrightarrow{D_j}) \times \text{idf}(t_i, D) \quad (3)$$

C. Feature Selection

We need to do the feature selection in our work for the following two reasons: First, we expect there have some features with similar values contained in the same type answers but differ between different types. These features provide a large amount of information for classification. At the same time, some features emerge frequently in all types. They need to be cancelled because they provide little information. Second, the dimension of our combined feature vector is more than one thousand, the more features, the larger possibility of overfitting for the training models.

We address both issues by seeking a compact, low dimensional and readable representation in feature selection. For this purpose, the information gain feature selection algorithm (InfoGain) [25] is chosen and used in our work. InfoGain evaluates the worth of an attribute by measuring the information gain with respect to the class. The higher the InfoGain for a feature means the more the information content the feature contains. Let T denote the training examples of the target type, each of the form $(\mathbf{x}, y) = (x_1, x_2, x_3, ..., x_k, y)$ where $x_a \in vals(a)$ is the value of the ath attribute of example $\mathbf{x}$ and y is the corresponding class label. The information gain for an attribute a is defined in terms of entropy $IG()$ as follows:

$$H(T) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) \quad (4)$$

$$H(T|a) = \sum_{v \in vals(a)} \frac{|\{\mathbf{x} \in T | x_a = v\}|}{|T|} \cdot H(\{\mathbf{x} \in T | x_a = v\}) \quad (5)$$

$$IG(T, a) = H(T) - H(T|a) \quad (6)$$

All the features are ranked according to the information gain metric. With the help of InfoGain algorithm, we can find the discriminative features from thousands of features.

D. Model Training

Model Training consists of classifying an answer document to its question's type. To achieve this objective, we consider both the absolute score and probability score for a given document. For the absolute score, the score of a specific document is just 0 or 1, 1 for belonging to its type, 0 for not belonging to its type; for the probability score, the score of a specific document is the range of [0,1], it reflects the degree of different documents belonging to a specific type. In detail, we adopt two algorithms: Bayesian Logistic Regression (BLR) [8] and Linear Regression(LR) [2]. [5]

Bayesian Logistic Regression: This is a good classifier for text classification. Genkin et al. [8] first proposed this approach for text categorization. This approach uses a Laplace prior to avoid overfitting and produces sparse predictive models for text data. So we adopt it to build the classification model in two ways:

1)Bayesian Logistic Regression with Threshold (BLRT): We assume that each answer document would just have two statuses, belonging to a specific type or not belonging to a specific type. The output set of BLRT is {0,1}. Let y be the output of a document d using BLR preditor. Given a threshold k, if y is larger than k, the document d would be labeled as 1; otherwise, d would be labeled as 0.

2)Bayesian Logistic Regression without Threshold (BLRnT): BLRnT is a variant of BLRT; it keeps the output y, which can be regarded as the probability of document d belonging to a specific type. The reason we adopt the BLRnT model is that, assuming there are two documents, d_i and d_j, that the document d_i is more likely to answer a positive type question than document d_j means that the probability of d_i belonging to a positive type is larger that d_j.

Linear Regression: LR is a linear approach to modeling the relationship between a scalar dependent variable y and variables $X = \{x_{i1}, ..., x_{ip}\}_{i=1}^{n}$. LR model assumes that the relationship between the dependent variable y_i and the p-vector x_i is linear. We adopt the LR model the same reason as BLRnT. The different between LR and BLRnT is, the LR assumes the relationship is linear, while BLRnT assumes non-linear.

To make the LR score comparable with BLRT and BLRnT score, we apply the equation 7 normalization on the LR score. Here m is the number of sample, $y_i(\mathrm{x})$ be the i^{th} output value by LR regression, and $\max_{j=1}^{m} y_j(\mathrm{x})$ is the maximum value, $\min_{j=1}^{m} y_j(\mathrm{x})$ is the minimum value. After normalization, LR score are in range [0.0, 1.0].

$$y_i(\mathrm{x}) = \frac{y_i(\mathrm{x}) - \min_{j=1}^{m} y_j(\mathrm{x})}{\max_{j=1}^{m} y_j(\mathrm{x}) - \min_{j=1}^{m} y_j(\mathrm{x})} \quad (7)$$

Based on these, we build classifiers for the six document sets, including "howto", "whether", "what", "why", "which" and "where". The main reason of six individual classifiers rather than a multi-classifier is that a document may be able to answer more than one type of questions. For example, a document may be not only good to answer "howto" questions but also "why" questions. In other words, the classification categories are not mutually exclusive.

[5]These two algorithms can be found in weka.jar. http://www.cs.waikato.ac.nz/ml/weka/downloading.html

V. Refinement in Software Text Retrieval

We build a text retrieval engine [36] to search for the relevant software documents for a given question. The search model is based on keyword matching. However, the precision is not very good because there are so many documents having similar keywords. To refine the search results, we merge our document classification step into repository building phase; these classification scores are used in question search phase to re-rank the search results.

To simulate a real text retrieval situation, we build a repository which stores all the answer documents. Each document in the repository is scored using the six classifiers we built. Thus each document has six scores to show the degree that it belongs to each type. It only needs to be run when the document is published to the repository. So it will not affect the response time of the search engine.

Once a question is submitted, the search engine transfers it into two parts: (1)One is to identify the interrogative tags. Our engine can identify the question's types by predefined regular expression patterns, which are similar to [14]. If the questioner is not satisfied with the question type auto-identification result, he/she could add or remove types. (2)The other part is query formulation, where the keywords are extracted for matching. The question "Using Lucene to count results in categories"[6] would be transformed to "use" + "lucene" + "count" + "result" + "categori" after the stopwords removal and stemming. Then the search engine returns a set of search results, where each document have a score described the similarity between the query and this document.

Once we get the candidate documents by text retrieval engine, the re-ranking approach is used to refine the search results. Some former works use the classifiers to filter out irrelevant documents (e.g. Gottipati et at. [9] use the classifier to filter out the junks). We name this re-ranking approach as "filter". There is another approach in our work, "rescoring", which is to re-score each document so as to get a new rank. Let x denote the feature vector of each document using the discriminative features, t be the type of the input question and $y()$ be the BLRT, BLRnT or LR method for estimating the possibility of document belonging to the question type. The re-score $s_R(\mathrm{x})$ is a linear combination of the original similarity $s_O(\mathrm{x})$ and the classifier score $y(\mathrm{x}, t)$, which is calculated as equation 8. The re-ranking is obtained by sorting s_R in descending order.

$$s_R(\mathrm{x}) = \alpha \cdot y(\mathrm{x}, t) + (1 - \alpha) \cdot s_O(\mathrm{x}) \tag{8}$$

VI. Experiments

To evaluate our approach, we have conducted a set of quantitative experiments. These experiments address the following questions:

RQ1:What affects the behaviour of the classifiers? The behavior of the classifiers has a great influence on the re-ranking approach. It would be affected by the feature weighting method and the feature set. Compared with our former work [36], we have increased the quantity of the learning examples and tried other feature weighting methods. We investigated their impacts individually to find the best classifier and the most representable features for each interrogative.

RQ2: How does our re-ranking approach work in software text retrieval? The objective of re-ranking is to improve the performance of software text retrieval. We are concerned about how well our re-ranking approach works. To evaluate it, we make some comparisons between our approach and existing text retrieval approaches. Also, we need to know the upper limit of our approach when the classifiers become perfect, i.e., achieving 100% in accuracy. The upper limit can be an indicator to instruct us in improving the classifiers to narrow the gab. Thus, we make a series of quantitative experiments to analyze the performance of our approach.

RQ3: How does our refinement approach work in different data set? We build our refinement models based on the question-answer pairs on *StackOverflow*. The learning examples are projects dependent and source dependent. How does it perform on other projects' text retrieval and other information coming from different web sources. To answer this question, we apply our approach to 7 famous open source projects, get their Frequently Asked Questions (FAQs) to investigate its performance in a large software text repository.

A. Dataset Setting Up

Our dataset is made up of three parts, which are obtained in different ways[7]:

Dataset-1: *Question-answer pairs on "Lucene" collected from StackOverflow.* As mentioned in paper [36], we first get 5,587 questions and 7,872 answers from the *StackOverflow* with tag "lucene", where **1,826** questions with positive votes are kept and labeled. We use these question and their **2,460** answers for original classifier training and testing.

Dataset-2: *Question-answer pairs on "Java" collected from StackOverflow.* We need more data to train the classifier models and evaluate our approach. Then we extend our data collection scope and randomly pick 50,000 questions with tag "Java" on *StackOverflow*. It may cost too much time if we judge the types of these question accurately and manually. We filter all the questions using regular expressions (e.g. the question includes phrases "how to" , "how can" or "what is the best way to", etc., are labeled with "how to" tag). Finally, **11,003** questions and the corresponding **16,255** answers are selected. Table IV briefly describes these two datasets.

TABLE IV: Dataset-1 and Dataset-2 for Evaluation

Tags	Collected with "Lucene"		Collected with "Java"	
	Questions	Answers	Questions	Answers
howto	1,134	1,494	5,655	7,774
whether	358	518	1,312	1,907
what	343	514	1,540	2,585
why	214	276	2,014	3,256
which	68	120	264	433
where	51	87	218	300
Total	1,826	2,460	11,003	16,255

[6]http://stackoverflow.com/questions/152127/using-lucene-to-count-results-in-categories

[7]Dataset for these experiments can be downloaded at: https://github.com/yetingqiaqia/L2R-Software-TR-QAs

Dataset-3: *FAQs of seven well-known open source projects.* In software development, FAQs are used by many projects as part of their documentation. Compared with the data from *StackOverflow*, the FAQs are more formal and accurate. We want to investigate whether our approach is valid in searching these questions' answers and whether the classifiers are affected by our learning examples. Table V illustrates the 7 open source projects and the numbers of their FAQs. All of them are the top level projects (TLPs) in Apache.

TABLE V: FAQs Dataset Description

Project	**Description**	**n. of FAQs**
Hadoop	A software framework that supports data intensive distributed applications	47
Hive	A warehouse software that manages large datasets residing in distributed storage	20
HTTP Server	An open-source HTTP server for modern operating systems	79
Lucene	A text search engine library	85
Maven	A project management and comprehension tool	23
Spark	A general engine for large-scale data processing	15
Tomcat	A web container for serving servlets and JSP	153

B. RQ1-Classifier Building

We use the Dataset-2 to train our classifiers. For a question type T, the positive samples are all the answer documents which are labeled with T, the negative samples are randomly sampled the same numbers as the positive samples from the answers of the other types. Then the learning examples are represented by feature vectors, including text features and non-text features. Finally, some important features are selected and the classifiers are built after feature selection and model training.

The ten-fold cross-validation method [2] is used to evaluate the classifiers. Thus, all samples are randomly split into ten parts. In each run, one part is used as test set, and the remaining nine parts are used as training set. The final evaluation results of each experiment is represented with the average of ten runs. And for the evaluation, the criteria of area under the curve (AUC) [7] is introduced to evaluate the accuracy of the classification.

ROC and AUC. The receiver operating characteristic (ROC) is a graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. A completely random guess would give a point along a diagonal line from the left bottom to the top right corners, for example, flipping coins (heads or tails). Figure 2 shows the ROC curve of BLRnT in dataset-2 using the TF-IDF feature weighting method.

The Area under the curve (AUC) is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. It is calculated by using an average of a number of trapezoidal approximations. Let P be the total number of positive instances, N be the total number of negative instances, TP be the number of true

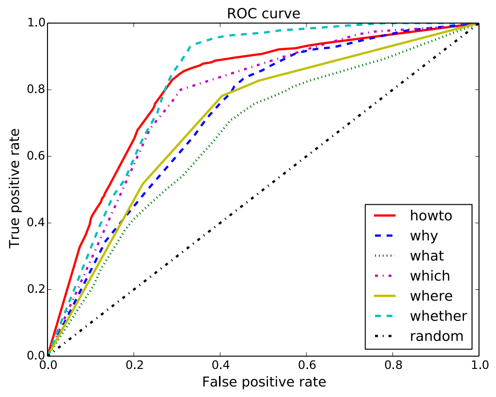

Fig. 2: ROC curve of TF-IDF

positive and FP be the number of false positive. Its calculation shows in equation 9.

$$AUC = \sum_{k=1}^{n} \frac{(FPR_k - FPR_{k-1})(TPR_k + TPR_{k-1})}{2} \quad (9)$$

$$TPR = TP/P, FPR = FP/N \quad (10)$$

For building our classifier, features are weighted using TF method or TF-IDF method. Table VI shows their effects to the classifiers. It could be noticed that the TF-IDF based features weighting method preforms better than TF method in five types document classification. For "howto" type documents, the classifier's AUC reaches 0.81 with TF-IDF based feature weighting method, which is 0.78 when TF based feature weighting method is used. At the same time, we can also see that the TF-IDF showed a bit falling than the TF in the document type of "whether".

TABLE VI: AUC scores using different feature weighting methods

Tags	**TF**	**TF-IDF**
howto	0.780	0.806
why	0.702	0.727
what	0.642	0.663
which	0.747	0.773
where	0.780	0.822
whether	0.776	0.741

Table VII shows the top-20 features learned from each answer document set, which are sorted based on their InfoGain scores. We can find that there are both text and non-text features, and interrogative types have different discriminative features. First, we analyze the text features. Some useful text features that have been used in previous approaches, e.g. "reason", "explain", "whi" (the root of "why"), "becaus" (the root of "because"), etc., are in [30], [31], and "first" in [38], [37]. And most other text features are very comprehensible, such as "yes" in *whether* type; "problem", "caus" (the root of "cause"), "mean", "issu" (the root of "issue"), "due" in *why* type; "choos" (the root of "choose"), "choic" (root of "choice"), "differ", "easier", "faster", "fastest" in *which* type;

TABLE VII: Top-20 Most Discriminative Features of Different Type Answers

Rank	howto		whether		why		what		which		where	
	Feature	Score	Feature	Score	Feature	Score	Feature	Score	Feature	Score	Feature	Score
1	Cod-avc	0.071	Cod-num	0.020	reason	0.024	Cod-mic	0.017	Cod-avc	0.010	download	0.037
2	Cod-mac	0.067	Cod-mac	0.016	problem	0.024	Cod-avc	0.016	Cod-mac	0.010	sourc	0.033
3	Cod-mic	0.059	yes	0.013	effect	0.022	Cod-mac	0.014	Cod-mic	0.010	locat	0.031
4	Cod-malc	0.051	Cod-avc	0.012	Len-sc	0.021	Cod-num	0.014	Cod-malc	0.009	find	0.025
5	Cod-num	0.047	Cod-malc	0.010	Cod-malc	0.021	Cod-milc	0.012	perform	0.009	file	0.022
6	Cod-milc	0.047	Cod-milc	0.010	Link-lc	0.020	Cod-malc	0.011	choos	0.008	directori	0.022
7	Cod-clc	0.020	http	0.008	caus	0.019	differ	0.007	good	0.004	Link-xlc	0.022
8	Cod-sdc	0.020	org	0.008	Cod-sdc	0.016	object	0.006	high	0.004	Link-lc	0.017
9	languag	0.019	github	0.008	case	0.015	languag	0.005	full	0.003	zip	0.015
10	Len-sc	0.015	think	0.008	whi	0.015	Len-sc	0.005	differ	0.003	src	0.013
11	Cod-sdlc	0.014	first	0.008	mean	0.013	instanc	0.005	opinion	0.003	tutori	0.012
12	framework	0.014	Cod-clc	0.007	becaus	0.012	librari	0.004	choic	0.003	db	0.012
13	implement	0.013	Len-sc	0.007	question	0.012	type	0.004	easier	0.003	api	0.011
14	Len-wc	0.012	differ	0.005	explain	0.012	perform	0.004	faster	0.003	book	0.011
15	make	0.010	search	0.005	decis	0.011	mean	0.004	popular	0.003	net	0.011
16	featur	0.009	basic	0.005	explicit	0.010	usual	0.004	fastest	0.002	svn	0.010
17	exampl	0.009	closest	0.004	issu	0.009	execut	0.004	recommend	0.002	project	0.010
18	first	0.008	releas	0.004	due	0.009	defin	0.003	requir	0.002	weblog	0.009
19	mean	0.008	support	0.004	situat	0.008	class	0.003	doubt	0.002	gnu	0.009
20	design	0.007	nope	0.004	argument	0.007	function	0.003	depend	0.002	Cod-num	0.009

"download", "sourc" (the root of "source"), "locat" (the root of "locate"), "file", "directori" (the root of "directory") "tutori"(the root of "tutorial"), "src", "svn", "book", "db" in *where* type.

Second, we take a look at those non-text features. Non-text features include code features, link features and length features. They all appear at the top 20. The code features (e.g. "Cod-num", "Cod-mac", "Cod-malc", "Cod-milc", etc.) have a good performance in *howto*, *whether*, *what* and *which* type. Meanwhile, link features (e.g. "Link-lc") have an important role in the *where* and *why* type. And the length features (e.g. "Len-sc") make some contribution to *howto*, *whether*, *why* and *what* type.

C. RQ2-ReRanking Evaluation

For the re-ranking evaluation, we use 1,826 questions in Dataset-1 to search in a software text repository composed of 28,294 answer documents. These answer documents in the repository include not only all the positively-voted answers in dataset-1 and dataset-2, but also the related zero-score answers and the negatively-voted answers. Since the ranking evaluation metric needs a rating assessment (i.e. degree of relevance) for each document of the given question, we calculate the degree of relevance with the following steps automatically: (1)For each question, we regard the corresponding positively-voted answers as the relevant answers while other answers are irrelevant. (2)The degree of relevance is measured by the score of the answer. We score the minimum positively-voted answer 1 and the second minimum answer 2, etc., while others are scored 0(irrelevant).

We introduce a commonly-used text retrieval evaluation criterion, **Normalized Discounted Cumulative Gain at top k (nDCG@k)** [15], to evaluate our re-ranking approach. nDCG@k allows us to measure how close the predicted answer ranking is to the ground truth ranking. More formally, it is defined as the following equation 11, where $DCG@k$ is a discount cumulative gain score of the question-answer relevance and IDCG@k is the maximum possible (ideal) $DCG@k$ score. It is defined as equation 12, where rel_i is the true rating assessment for the answer at positon i in the ranking.

$$nDCG@k = \frac{DCG@k}{IDCG@k} \tag{11}$$

$$DCG@k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{12}$$

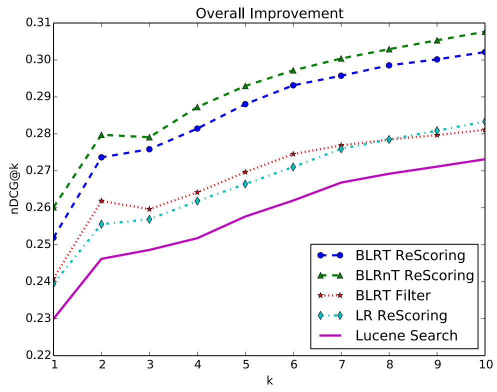

Fig. 3: Improvement of Different Methods on Dataset-1 (nDCG@k)

Figure 3 shows the entire improvement. In the overall improvement, the filter performs worse than BLRT re-scoring and BLRnT re-scoring. And the filter behaves unstably. LR re-scoring performs worse among all the re-ranking methods. BLRnT based re-scoring approach obtains an improvement of 13.1% in nDCG@1, 13.6% in nDCG@2 and 12.6% in nDCG@10 upon the baseline. BLRT re-scoring approach obtains an improvement of 9.5% in nDCG@1, 11.1% in

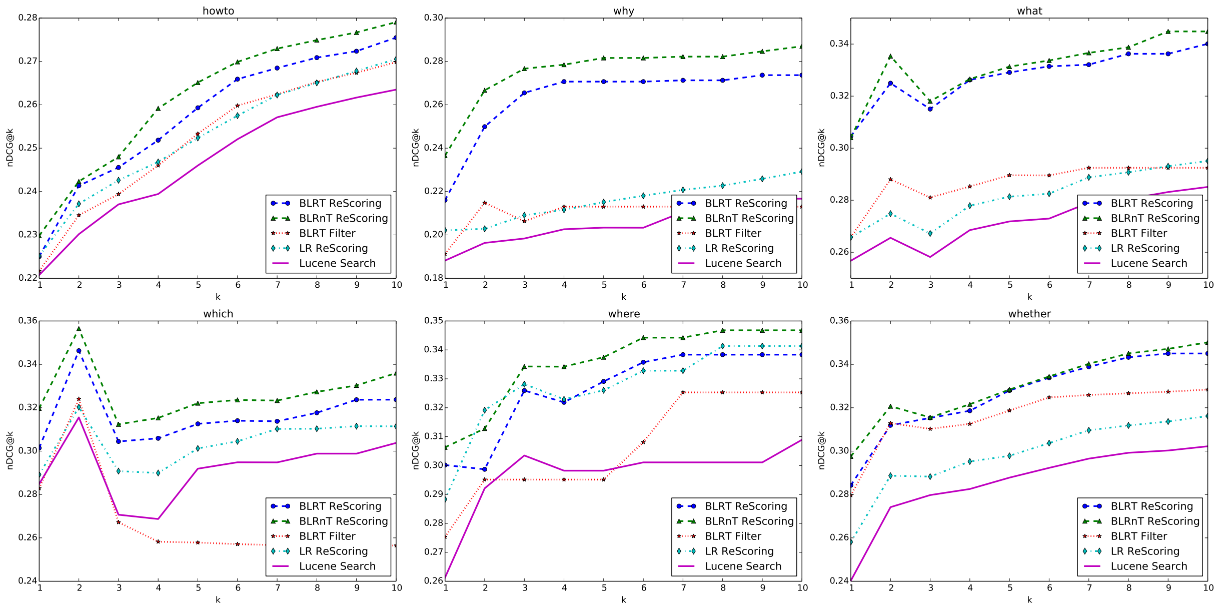

Fig. 4: Improvement of Different Methods on the DataSet-1 per Question Type (nDCG@k)

nDCG@2 and 10.6% in nDCG@10. The BLRnT re-scoring outperformes the other methods for all values of k (nDCG@k) in the overall improvement in dataset-1. Figure 4 shows the nDCG@k evaluation results of different question types. Since our work addresses the problem of ranking answers for software text retrieval, we set the Lucene Search as the baseline. We compare with the filter approach (Gottipati et al. [9]) and the LR re-scoring approach. We observe that almost all the re-ranking methods outperformed the baseline (Lucene Search) in all question types. This result indicates that almost all the re-ranking methods show a better performance than baseline.

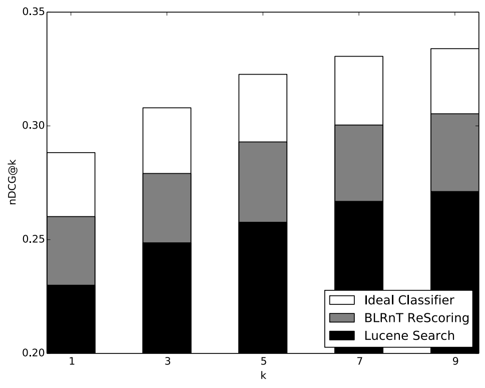

Fig. 5: Comparison of Ideal Approach, BLRnT Re-Scoring and Lucene Search

Since our classifiers aren't perfect, the nDCG@k score of our re-ranking method would still have some distance with the upper limit when the classifiers are perfect (i.e. all the answers are classified correctly). To illustrate the gap visually, we build a virtual perfect classifier which can classify all the answers correctly. Then we apply this classifier to re-rank the documents. The overall comparison between Ideal Approach, BLRnT Re-Scoring and Lucene Search is illustrated in Figure 5. The BLRnT re-scoring obtains the best nDCG@1 of 26.1% ($\alpha = 0.13$) and the ideal approach obtains the best nDCG@1 of 28.8% ($\alpha = 0.15$). It indicates that there still has some room for our approach to improve.

D. *RQ3-Application Evaluation*

We investigate whether our refinement approach and the trained classifier models could be applied to other software resources. So to answer RQ3, we collect 7 well-known open source projects' FAQs from their websites. Then we mix the FAQ's answers with Dataset-1 and Dataset-2 to simulate a context of software text retrieval. Once a FAQ question is input to our search engine, the Lucene search engine would return the top-30 matched documents, then the BLRnT scores would be added using the equation 8. Our evaluation focuses on whether our re-rank approach could re-rank its answer to the top of the search result.

Figure 6 shows the improvement of our methods over the baseline of keywords search (Lucene Search). In the best case, the BLRnT re-scoring method gets 12.6% improvement in nDCG@1 and 7.9% improvement in nDCG@10. And the BLRT re-scoring method gets 13.2% improvement in nDCG@1 and 7.1% improvement in nDCG@10. The evaluation is based on all the 422 FAQ questions' search and retrieval. It indicates that our classifier could be used effectively in large software text repositories.

We compare the ranking results between BLRnT re-scoring and Lucene search project by project in Table VIII. Here **RB**↑ represents the number of FAQs Ranked Better after re-scoring; **RW**↓ represents the number of FAQs Ranked Worse after re-scoring; **RT1**↔ represents the number of FAQs Ranked at

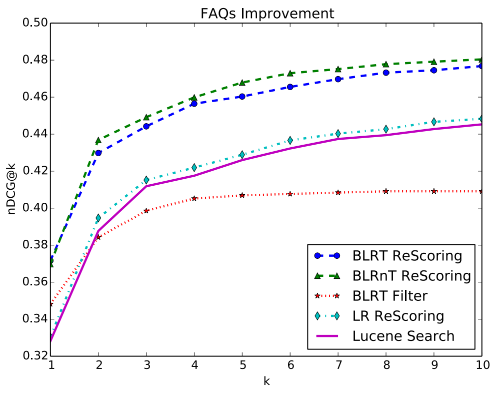

Fig. 6: nDCG@k Evaluation of Our Approach on DataSet-3

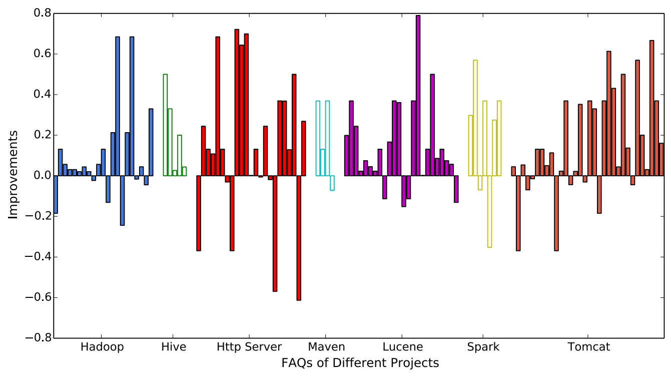

Fig. 7: The Detail of RB↑ and RW↓ in Experiments

Top 1 both at the Lucene Search step and the BLRnT re-scoring step; **RnT1**↔ represents the number of FAQs Ranked the same but Not Top 1 both at the Lucene Search step and the BLRnT re-scoring step; **NA** represents the number of FAQs Not Appeared at the top-30 matched documents of the search engine, so this answer will not be re-scored by BLRnT either. We can see that (1) 88 (20.9%) search results turn good and only 28 (6.6%) search results turn bad; (2) 138 (32.7%) questions' best answers rank at top 1 both at the Lucene Search step and the BLRnT re-scoring step, these answers would not be re-scored better by BLRnT; (3) at the same time, we also find that 146 (34.6%) search results could not return the input question's answer in the candidate list. That is mainly because there are too many software documents having the same keywords as the input question. We will pay more attention to keyword matching methods in the future.

TABLE VIII: Ranking Result Comparison between BLRnT Rescoring and Lucene Search on Different Projects' FAQs

Project	**RB↑**	**RW↓**	**RT1↔**	**RnT1↔**	**NA**	**Total**
Hadoop	15	6	10	3	13	47
Hive	5	0	6	0	9	20
Http Server	16	7	32	2	22	79
Maven	3	1	6	2	11	23
Lucene	20	4	21	7	33	85
Spark	5	2	5	1	2	15
Tomcat	24	8	58	7	56	153
SUM	88	28	138	22	146	422

We plot the search results of **RB**↑ and **RW**↓ further. For every search result, the improvement value is the question's nDCG@10 score after re-scoring minus its original nDCG@10 score. It is illustrated in Figure 7. Here we can see that the changes on the positive (turn good) situations are obviously more than the negative (turn bad) situations. The performance shows no big difference among different projects' results. So, the improvement is project independent.

E. Threats to Validity

There are a number of threats to the validity of our results.

1) Threats to internal validity corresponds to the relationship between the independent variables (i.e., input variables of document features) and dependent variables(i.e., target variable) in the study. Our work is based on supervised machine learning, the quality and quantity of learning examples are very important. We have tried to minimize this error by performing cross-validation on the manually created labels. And we only chose the positively voted question answer pairs for evaluation from *StackOverflow* and increased the dataset by adding the "Java" tag question-answers and FAQs. The bias is minimized by these efforts.

2) Threats to external validity corresponds to the generalizability of the result. For training and evaluating the classifiers, we have experimented with 16,255 answers using ten-fold cross-validation. We simulated a large software text retrieval scenario by mixing 28,294 answer documents together. Then we used 1,826 "lucene" tag questions and 422 questions from projects' FAQs for evaluating the re-ranking approach.

3) Threats to construct validity corresponds to the appropriateness of our evaluation metrics. We used standard metrics: AUC for evaluating the classifiers and the nDCG@k for re-ranking performance. Comparing with precision, recall and F-score metric in our previous work [36], the AUC metric performs more stably for it does not take threshold into account. nDCG is a widely used metric for ranking [33]. By enforcing fixed set size k for the result set and using minimum score 0 (or negative score) for the bad documents and the missing documents, the nDCG@k can penalize for bad documents and missing documents. Thus, we believe there is little threat to construct validity.

VII. Discussion and Related Work

Interrogatives have received much attention in many text retrieval tasks. For example, Yin et al. [38], [37] address the problem of automatically retrieving answers for "how-to" questions. They target six procedural elements, like *first*, *second*, and proposed a two-stage approach to retrieving answers for "how-to" questions. Tanaka et al. [27] , Verberne et al. [30], [31] and Higashinaka et al. [12] also have done a lot of work to deal with "why" questions. Different from their works, we try to better retrieve answers for questions of all common interrogatives but focus on programming questions. The answers of programming questions differ from other open domain questions' answers. The words and phrases used will be restricted; code snippets and links appear frequently. With these differences, it is necessary to train new classifiers for software documents.

The basic idea behind our approach is to build a document classifier in software engineering. It learns from question-answer pairs on *StackOverflow* and selects discriminative features to train a better classification model. In the process, we learn that some non-text features are also very important in software document classification, such as code features in answering questions of *howto*. It indicates that the number of code lines is a very important factor in software engineering documents. On the other hand, several features show less importance than our expectation according to the observation of our experiment results, such as sequence words(e.g. first, second, third, etc.) for *howto* type. For example, in the sentence "I prefer the first method", the word "first" represents a choice for answering the *which* question. In this example, the word "first" does not contribute to the *howto* question. Non-text features are also considered in other software text retrieval works. For example, Bacchelli et al. [1] present an approach based on island parsing to recognize the structured information; Hill et al. [13] add keyword's location information to calculate relevance; Yuan et al. [28] use the code facts features to identify the Linux bug fixing patches.

In our work, questions are labeled in advance. Question classification is very hot research topic in information retrieval [14], [29]. Treude et al. [29] identify ten main question types: how-to, discrepancy, environment, error, decision help, conceptual, review, non-functional, novice, and noise. Our question types are set up from different search focuses – interrogatives. Different interrogatives need different answers. In practice, the interrogative phrases should be right parsed and understood. For example, given two questions: (1) "What is StandardAnalyzer?" (2)"What's the best way to call StandardAnalyzer?" They all contain "what", but they have different answer requirement. Considering these factors, we pick a series of regular expressions for all the question types to help a better match. In the future, some automatic or semi-automatic question classification approach will be studied and included in our work.

A lot of rank assistant approaches have been proposed to improve the performance of software text retrieval [15]. For example, wang et al. [32] incorporate users' opinion on the results from a code search engine to refine result lists and gains 11.3% comparing with Portfolio, the best performing code search engine at that time. Changsheng et al. [19] search and analyze the software comments on the Internet so as to find more candidate results with related quality features. Leelaphattarakij et al. [17] present a method to rank application search results focusing on two criteria: application attributes and application context. Using application attributes and its context, they propose three ranking scores: rating score, content relevant score and context score. Kimmig et al. [16] give an approach for translating natural language queries to concrete parameters of a third party code query engine. Haiduc et al. [10] give a query reformulation strategy to improve the performance of text retrieval. Instead of that developers have to map their questions to multiple concrete queries that can be answered, Wursch et al. [34] present a framework to query for information about a software system using guided-input natural language resembling plain English. To deal with so many features that maybe affect text ranking, researchers of Microsoft Liu et al. [20] point out that machine learning should be used to mine those discriminative features and re-rank the retrieval results.

Our dataset comes from *StackOverflow*. It provides plenty of programming question and answer pairs and gives us a way to learn what the answer would be like when a question is submitted. A lot of work has been done to study the relationship between software questions and their relevant answers. Some researches focus on the quality of question and answer pairs [5], [9], [23], [35]. These work focus on ranking the answers exactly related to the given question(i.e. the answers which answer the same question). Different from them, we are interested in ranking all the retrieved documents to the given question.

VIII. Conclusion and Future Work

In this paper, we present an interrogative-guided re-ranking approach for question oriented software text retrieval. We build several software document classifiers, which learn from more than 16,000 question-answer pairs on *StackOverflow*. Based on these classifiers, we select top-20 discriminative features from answers to 6 type questions: *howto*, *whether*, *what*, *why*, *which* and *where*. Then, we apply these classifiers to software text retrieval, re-scoring every search result by combining its classification score and its text retrieval score. In the experiments, our re-ranking approach presents 13.1% improvement in nDCG@1 upon the baseline and 12.6% improvement in nDCG@10 respectively. We also apply our approach to 7 open source projects. The results of our experiments suggest that our approach could find FAQs' answers more precisely. Also, the improvement is project independent and source independent.

As for future work, to improve the effectiveness of our proposed approach further, we plan to improve the classification accuracy of our classifiers further and apply this approach to more real software text repositories. First, the learning data should be labeled faster and more accurately. We plan to combine the regular expression matching method and part-of-speech tagging method together to make question labeling automatically as well as ensuring the quality of the data. Second, the performance of our classifier still needs further improvement. Our classifier's AUC can reach 0.822 now in the best case, so there is room for further improvement. We propose to rethink the features we use and try more classification methods to gain better performance. Third, we simulated the large software text retrieval scenario by mixing the FAQs' answer with all other documents, which assumes there must be answers in the repository. However, in application, there are all kinds of data in software repository, which may affect the search performance. We also plan to carry more experiments in real software text repositories and perform a user study to evaluate the effectiveness of our approach in refining the search results of programming questions.

IX. Acknowledgments

This research is sponsored by the National Basic Research Program of China (973) under Grant No. 2015CB352201, the National Natural Science Foundation of China under Grant Nos. 61421091, 61432001, 61472007.

References

[1] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci. Extracting structured data from natural language documents with island parsing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 476–479. IEEE, 2011.

[2] C. M. Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[3] S. Cai, Y. Zou, L. Wang, B. Xie, and W. Shao. A semi-supervised approach for component recommendation based on citations. In *ICSR*, pages 78–86. Springer, 2011.

[4] Y. C. Cavalcanti, I. d. C. Machado, P. A. Neto, E. S. de Almeida, and S. R. d. L. Meira. Combining rule-based and information retrieval techniques to assign software change requests. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 325–330. ACM, 2014.

[5] D. H. Dalip, M. A. Gonçalves, M. Cristo, and P. Calado. Exploiting user feedback to learn to rank answers in q&a forums: a case study with stack overflow. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 543–552. ACM, 2013.

[6] H. Drucker, S. Wu, and V. N. Vapnik. Support vector machines for spam categorization. *Neural Networks, IEEE Transactions on*, 10(5):1048–1054, 1999.

[7] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[8] A. Genkin, D. D. Lewis, and D. Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.

[9] S. Gottipati, D. Lo, and J. Jiang. Finding relevant answers in software forums. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 323–332. IEEE Computer Society, 2011.

[10] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[11] F. M. Harper, D. Moy, and J. A. Konstan. Facts or friends?: distinguishing informational and conversational questions in social q&a sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 759–768. ACM, 2009.

[12] R. Higashinaka and H. Isozaki. Corpus-based question answering for why-questions. In *IJCNLP*, pages 418–425, 2008.

[13] E. Hill, L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527. IEEE Computer Society, 2011.

[14] Z. Huang, M. Thint, and Z. Qin. Question classification using head words and their hypernyms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 927–936. Association for Computational Linguistics, 2008.

[15] K. Järvelin and J. Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 41–48. ACM, 2000.

[16] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 376–379. IEEE Computer Society, 2011.

[17] A. Leelaphattarakij and N. Prompoon. Ranking application software retrieval results using application attributes and context. In *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on*, pages 231–235. IEEE, 2012.

[18] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[19] C. Liu, Y. Zou, S. Cai, B. Xie, and H. Mei. Finding the merits and drawbacks of software resources from comments. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 432–435. IEEE Computer Society, 2011.

[20] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[21] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[22] A. Marcus and G. Antoniol. On the use of text retrieval techniques in software engineering. In *Proceedings of 34th IEEE/ACM International Conference on Software Engineering, Technical Briefing*, 2012.

[23] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 25–34. IEEE, 2012.

[24] M. F. Porter. Snowball: A language for stemming algorithms, 2001.

[25] D. Roobaert, G. Karakoulas, and N. V. Chawla. Information gain, correlation and support vector machines. In *Feature Extraction*, pages 463–470. Springer, 2006.

[26] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):26, 2014.

[27] K. Tanaka, T. Takiguchi, and Y. Ariki. Towards domain independent why text segment classification based on bag of function words. In *AI 2012: Advances in Artificial Intelligence*, pages 469–480. Springer, 2012.

[28] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 386–396. IEEE, 2012.

[29] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web?: Nier track. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 804–807. IEEE, 2011.

[30] S. Verberne, L. Boves, N. Oostdijk, and P.-A. Coppen. What is not in the bag of words for why-qa? *Computational Linguistics*, 36(2):229–245, 2010.

[31] S. Verberne, H. van Halteren, D. Theijssen, S. Raaijmakers, and L. Boves. Learning to rank for why-question answering. *Information Retrieval*, 14(2):107–132, 2011.

[32] S. Wang, D. Lo, and L. Jiang. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 677–682. ACM, 2014.

[33] Y. Wang, L. Wang, Y. Li, D. He, W. Chen, and T.-Y. Liu. A theoretical analysis of ndcg ranking measures. In *Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013)*, 2013.

[34] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 165–174. ACM, 2010.

[35] Y. Yao, H. Tong, T. Xie, L. Akoglu, F. Xu, and J. Lu. Want a good answer? ask a good question first! *arXiv preprint arXiv:1311.6876*, 2013.

[36] T. Ye, B. Xie, Y. Zou, and X. Chen. Interrogative-guided re-ranking for question-oriented software text retrieval. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 115–120. ACM, 2014.

[37] L. Yin. A two-stage approach to retrieving answers for how-to questions. In *Proceedings of the Eleventh Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 63–70. Association for Computational Linguistics, 2006.

[38] L. Yin and R. Power. Adapting the naive bayes classifier to rank procedural texts. In *Advances in Information Retrieval*, pages 179–190. Springer, 2006.

[39] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

[40] Y. Zou, L. Zhang, Y. Li, B. Xie, and H. Mei. Result refinement in web services retrieval based on multiple instances learning. *Web Services Research for Emerging Applications: Discoveries and Trends: Discoveries and Trends*, page 340, 2010.