Query-Based Configuration of Text Retrieval Solutions for Software Engineering Tasks

Laura Moreno[1], Gabriele Bavota[2], Sonia Haiduc[3], Massimiliano Di Penta[4], Rocco Oliveto[5], Barbara Russo[2], Andrian Marcus[1]
[1]The University of Texas at Dallas, Richardson, TX, USA
[2]Free University of Bozen-Bolzano, Bolzano, Italy; [3]Florida State University, Tallahassee, FL, USA
[4]University of Sannio, Benevento, Italy; [5]University of Molise, Pesche (IS), Italy

ABSTRACT

Text Retrieval (TR) approaches have been used to leverage the textual information contained in software artifacts to address a multitude of software engineering (SE) tasks. However, TR approaches need to be configured properly in order to lead to good results. Current approaches for automatic TR configuration in SE configure a single TR approach and then use it for all possible queries. In this paper, we show that such a configuration strategy leads to suboptimal results, and propose QUEST, the first approach bringing TR configuration selection to the query level. QUEST recommends the best TR configuration for a given query, based on a supervised learning approach that determines the TR configuration that performs the best for each query according to its properties. We evaluated QUEST in the context of feature and bug localization, using a data set with more than 1,000 queries. We found that QUEST is able to recommend one of the top three TR configurations for a query with a 69% accuracy, on average. We compared the results obtained with the configurations recommended by QUEST for every query with those obtained using a single TR configuration for all queries in a system and in the entire data set. We found that using QUEST we obtain better results than with any of the considered TR configurations.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Documentation, Measurement

Keywords

Text-Retrieval in Software Engineering, Configuration, Feature and Bug Localization

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786859

1. INTRODUCTION

Software systems contain a large amount of information in text format, captured in a multitude of software artifacts, such as, requirements documents, design documentation, source code, bug reports, developer communication, user manuals, test documents, *etc.* This information, when properly utilized, can help developers understand various aspects of a software system and can help them in their daily tasks. Text Retrieval (TR) approaches have been proposed to leverage the large amount of text information available in software. These techniques have been successfully applied to support more than 20 different software engineering tasks [37], such as feature and bug localization in source code [38], traceability link recovery [4], impact analysis [11], bug triaging [5], and so on.

The research on using TR in software engineering (SE) has generated several hundred papers in the last decade, with a focus on investigating how to support specific tasks and how to improve the performance of specific TR approaches. Most of the published work is empirical in nature, and many tools and research prototypes have been produced. TR techniques need to be configured before using them, which requires calibrating several parameters. Different parameter values lead to different TR configurations, and the work on determining the best TR configurations in SE [35, 40, 48] reached the conclusion that the performance of a TR configuration varies widely from system to system, even when used for the same task. More than that, a TR configuration that works best for one software corpus may not be the best for another.

Different automated approaches have been proposed to calibrate some TR techniques for specific SE tasks and for individual software corpora, either based on specific heuristics [6, 22, 48], or by relying on search-based optimization techniques [35, 40]. While this previous work has acknowledged the need to adapt the configuration of the TR techniques to the characteristics of each individual software corpus and SE task, it has overlooked one aspect: *the query.*

Other previous work showed that, while some queries lead to good results, others fail to retrieve any useful information when using a specific TR configuration. This is mainly because different queries have different lexical and semantic properties [12], which may require different approaches for finding relevant documents. In consequence, researchers also focused on (automatically or manually) changing the queries [19, 24] to fit a given TR configuration and software corpus.

Based on all existing work, we argue that there is no "silver bullet" TR configuration that can work equally well for all queries formulated in the context of a SE task and a given software corpus. In order to verify such assumption, we conducted an exploratory study and reported the performance of 21 TR configurations used in previous SE work, using a large data set of software systems and queries for the task of feature and bug localization. This study (see Section 3) confirmed our conjecture that no single TR configuration works best for all queries in a system and that the difference in performance between TR configurations varies greatly across queries in the same system. In other words, these findings indicate that the query is an important factor in determining the best TR results for a given software corpus.

In this paper, we tackle the TR configuration problem at a new granularity level, *i.e.,* the query level. We propose a new approach, named QUEST (**QUE**ry-based configuration for **S**oftware re**T**rieval), which considers both the software corpus and the query to automatically determine the best TR configuration to use for each individual query in the context of a given SE task. QUEST relies on supervised learning, and uses a training set of queries and their relevant results for building a classification model. For each query, QUEST computes a set of measures that capture different properties of the query that have been shown to be useful in other TR tasks, such as, query quality prediction [25] and query reformulation [24]. The learning process relies on automatically running a set of queries using different TR configurations on labeled training data and determining which configuration performs the best for each query according to its properties. When a new query is issued, the model is then used to automatically determine the best TR configuration to use for the query, based on its properties.

We evaluated QUEST in an empirical study on feature and bug localization data. Part of the evaluation focused on determining how accurate is QUEST in determining the best configuration for a given query. We found that QUEST is able to recommend one of the top three TR configurations for a query with a 69% accuracy, on average. The evaluation also focused on determining whether the configuration selected by QUEST for each query leads to better results (for feature and bug localization) than running any of the individual TR configurations on all the queries. We found that for 76% of the queries (on average) the configuration selected by QUEST improves or preserves the quality of the retrieval.

Novel contribution. While previous work attempted to find the best TR configuration for a given software corpus and/or for a specific task [6, 22, 48, 35, 40] or to reformulate a query to fit a given TR configuration [19, 24], QUEST is the first technique that determines the best TR configuration for an individual query and a given software corpus.

Replication package. A replication package is available online[1]. This package includes (i) data sets used in our studies; (ii) complete results of the exploratory study; (iii) the definition of the query property measures currently implemented by QUEST; and (iv) complete results and analysis of the empirical study.

2. RELATED WORK

Hundreds of approaches have used TR techniques to address software engineering tasks. The vast majority of such approaches have determined the TR configurations to use based on previous work in natural language processing, or by relying on ad-hoc decisions [3, 15, 34, 38]. The most relevant work to this paper deals, however, with comparing different TR configurations for various software engineering tasks or automatically determining the best TR configuration for a particular task and data set. We discuss each of these two research directions below. No prior work has addressed TR configuration at query level in software engineering.

Several studies have compared TR configurations in order to determine the most suited one for a specific software engineering task. Abadi *et al.* [1] compared five TR engines and several parameter values for each engine for the task of traceability link recovery and found Lucene and Jensen-Shannon as the best engines to use. This same task was addressed by Gethers *et al.* [21], which combined orthogonal TR approaches for achieving better results. Falessi *et al.* [17] compared a large number of TR configurations and combinations of TR engines for discovering equivalent requirements. Shi *et al.* [46] used BM25 and BM25F with several fixed parameter configurations and compared them with other TR approaches for feature location, showing that the BM25 approaches performed the best. Rao *et al.* [44] addressed bug localization and compared five TR engines using different parameter configurations, as well as combinations of two TR engines. They found that the Unigram model and its respective composite approach performed the best. Thomas *et al.* [48] also considered bug localization for their large scale comparison of TR configurations; over 3,000 configurations were tested and combinations of TR engines were also considered. The study showed that the best individual TR engine is the Vector Space Model (VSM), and that combinations of TR are always beneficial no matter which TR models are combined. Wang *et al.* [49] followed this direction and used a genetic algorithm in order to determine a near-optimal composition model for several VSM variants for bug localization. Lukins *et al.* [36] and Nguyen *et al.* [39] addressed Latent Dirichlet Allocation (LDA) configuration for bug localization, comparing it with other approaches.

Most of the studies mentioned so far focus on finding one TR configuration or a combination of TR configurations that achieve the best performance across all data sets, as a general prescription to be used by researchers for similar tasks. However, one of the most outstanding and frequent lessons of these studies is that different TR configurations may be needed for different data sets. As an illustration, Wong *et al.* [50], who used VSM for bug localization, found that the optimal parameter settings are different from project to project. Biggers *et al.* [6], focusing on the configuration of LDA for the task of feature location, showed that different parameter values are needed for systems having different sizes. Binkley *et al.* [7] emphasized this point and studied the impact of LDA parameter configuration on results. The study underlines the fact that there is no universal best setting for LDA parameters and that the optimal settings depend on the SE task, the input corpus, and the information needs of the developer. As a result, some approaches were proposed for determining the best TR configuration for a particular data set. Kuhn *et al.* [31] proposed a formula for determining the number of Latent Semantic Indexing (LSI) dimensions for a system based on its corpus size. On a similar note, Grant *et al.* [23] devised an approach to determine the number of topics to use in LDA for SE applications based on the

[1]http://www.utdallas.edu/~lmorenoc/research/fse2015-quest/

Table 1: Data set used in the studies.

System	Version	KLOC	# of methods	# of issues	# of queries
apache-nutch	1.8	50	2,230	10	30
apache-nutch	2.1	43	2,114	18	54
bookkeeper	4.1.0	52	3,382	27	81
commons-math3	3.0	197	9,173	16	48
derby	10.9.1.0	1,020	41,576	33	99
mahout	0.8	172	9,485	22	66
openjpa	2.0.1	508	41,911	22	66
pig	0.8.0	311	15,095	35	105
pig	0.11.1	440	19,478	55	165
solr	4.4.0	834	30,982	37	111
tika	1.3	69	3,401	20	60
zookeeper	3.4.5	85	4,711	50	150
Total	*12*	*3,782*	*183,538*	*345*	*1,035*

number of code fragments in a system. Recent approaches made use of genetic algorithms in order to determine the best parameter values to use with a given data set for LDA (see Panichella *et al.* [40]), as well as VSM and LSI (see Lohar *et al.* [35]).

A parallel set of approaches called "learning-to-rank" (LtR) has been studied for retrieving relevant software artifacts for different SE tasks. Binkley and Lawrie [8] studied LtR in the context of traceability link recovery and feature location, and Ye *et al.* [52] used it for bug localization. LtR approaches, which were first introduced in the last decade in the field of Information Retrieval [32], make use of machine learning in order to learn the rank of each document in response to a given query [33] or set of queries [20, 30]. These approaches use a set of features for each *(query, document)* pair, which can indicate some properties of the query and document, including the similarity of the query to the document based on some TR approach. However, when TR approaches are used to obtain such features, their parameters are set to the same values for all queries. In LtR approaches, the final list of results for a query is not given by a TR approach therefore LtR does not deal with TR configuration. Moreover, LtR builds a model that is trained for ranking documents and then the same model is used to rank the documents for all queries. In our approach, we use different TR configurations depending on the query. LtR approaches are also very computationally intensive, as they require learning the rank of each individual document with respect to each query, making these models very complex and impractical for large corpora [32]. Our approach requires learning only the TR configuration to use for a query, which can be applied to all documents, as opposed to learning the rank of every document in the corpus.

3. EXPLORATORY STUDY

Is there a single TR configuration that works best for all queries, given a software corpus? We performed an exploratory study to answer this question, in the context of feature and bug localization.

3.1 Data Set

We performed the study on a set of 12 versions of ten software systems, which are maintained by the Apache Software Foundation[2], belong to various domains, and have various sizes (see Table 1). In feature and bug localization studies using TR approaches, it is common to use the issue tracking system and the versioning system of a software in order to extract the evaluation data. The queries for a system are extracted from change requests (*i.e.,bug reports* and *feature requests*) found in the issue tracker of that system. Each change request has a title and a description, both containing textual information which describes the problem (in the case of bug reports) or the feature to be implemented (in the case of feature requests). Queries in TR studies are usually extracted from these titles and descriptions [16].

In our study, we extracted three representative queries for each change request: one from its title, one from its description, and one merging the title and the description. Therefore, for the 345 issues considered in the study, we obtained a total of 1,035 queries. After extracting the queries, we applied common processing techniques to the text in the queries, such as, splitting (*i.e.,* separating composed words, such as identifiers, into individual terms), filtering (*i.e.,* removing common English words and programming keywords), and stemming (*i.e.,* mapping words to their lexical root). We used the Porter stemmer [43] and a stop word list that is available in the replication package.

The *gold set* (*a.k.a.* ground truth) for the evaluation is formed by the set of changed methods in response to each bug report or feature request. So, we focused on bug reports and feature requests that have been successfully resolved, *i.e.,* those marked as $resolution = \text{Fixed}$ and $status \in \{\text{Resolved}, \text{Closed}\}$. We extracted the modified methods from the commits that implemented the change requests, found in the versioning system of each software. In particular, we linked each resolved issue to the commit pushing it in the versioning system by using the approach proposed by Fischer *et al.* [18], which relies on matching the issue ID in the commit note (*e.g.,* "fix commit #ID", "fixing #ID"). All the systems used in our study adopt this convention for keeping traceability between commits and issues.

For each software system, we built a text corpus from its source code by considering each method as a separate document and extracting the identifiers and comments of each the method. This corpus was then processed using the same sequence of techniques applied to the queries: splitting, filtering, and stemming. The corpus for each system was indexed using each TR configuration, and used to retrieve the relevant methods in a system, based on the queries.

3.2 Methodology

We applied 21 different TR configurations (see Table 2) to the queries in the data set and analyzed their results. The TR configurations were selected by considering the ones rec-

[2]`http://www.apache.org`

Table 2: TR configurations used in the studies.

Engine	Configuration	ID
Lucene [48]	Default	E1C1
LDA	Variable according to the system size [6]	E2C1
	$n = 500$, $iterations = 500$, $\alpha = 0.5$, $\beta = 0.5$	E2C2
	$n = 400$, $iterations = 200$, $\alpha = 0.5$, $\beta = 0.25$	E2C3
	$n = 300$, $iterations = 300$, $\alpha = 0.1$, $\beta = 1$	E2C4
BM25 [46]	$k_1 = 1.2$, $b = 0.5$	E3C1
	$k_1 = 1.2$, $b = 0.75$	E3C2
	$k_1 = 1.5$, $b = 0.5$	E3C3
LMJM [46]	$\lambda = 0.1$	E4C1
	$\lambda = 0.5$	E4C2
	$\lambda = 0.7$	E4C3
LMD	Default	E5C1
DFR	$model$ = Bose-Einstein, $after\text{-}effect$ = Bernoulli, $norm$ = H1	E6C1
	$model$ = Poisson-based tf-idf, $after\text{-}effect$ = Laplace, $norm$ = H1	E6C2
	$model$ = Poisson-based tf-idf, $after\text{-}effect$ = Laplace, $norm$ = H2	E6C3
	$model$ = Poisson-based tf-idf, $after\text{-}effect$ = Bernoulli, $norm$ = H1	E6C4
	$model$ = Poisson-based tf-idf, $after\text{-}effect$ = Bernoulli, $norm$ = H2	E6C5
IBM	$distribution$ = log-logistic, λ = df, $norm$ = No	E7C1
	$distribution$ = log-logistic, λ = ttf, $norm$ = No	E7C2
	$distribution$ = smoothed power-law, λ = df, $norm$ = No	E7C3
	$distribution$ = smoothed power-law, λ = ttf, $norm$ = No	E7C4

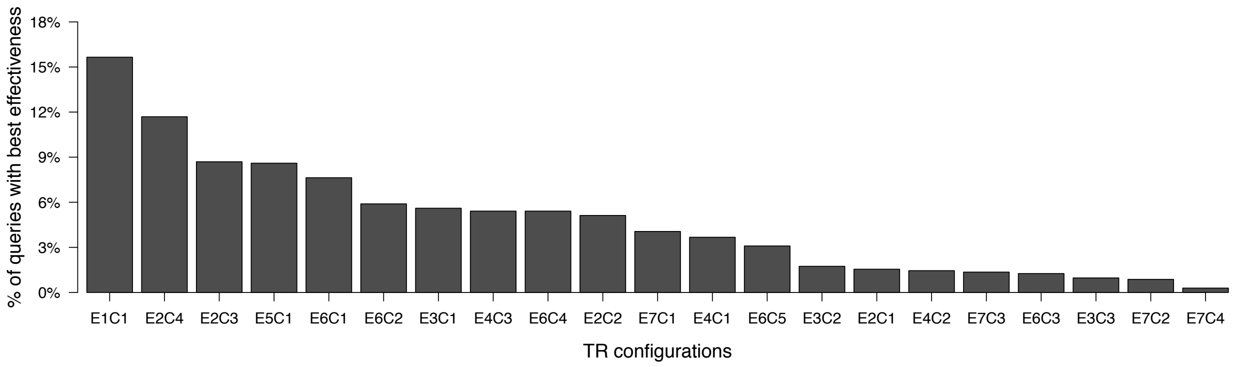

Figure 1: Percentage of queries for which the TR configurations obtain the the best effectiveness.

ommended in previous work focusing on TR tuning in SE [6, 40, 48]. In addition, we also considered some new TR configurations, which were not used before in the context of SE applications. The 21 TR configurations correspond to seven different TR engines: Lucene, LDA [9], Okapi BM25 [45], Language Model [42] with Jelinek-Mercer smoothing (LMJM) [54], Language Model with Dirichlet smoothing (LMD) [54], Divergence From Randomness (DFR) [2], and Information-Based Model (IBM) [13].

We evaluated the performance of the TR configurations by measuring the *effectiveness* of the retrieval for each query in the data set. *Effectiveness* is a measure widely used for evaluating TR approaches for feature and bug localization [16]. It represents the position of the first relevant document in the list of results, therefore being a proxy of the number of documents a developer would have to look at before finding the first relevant answer to the query (*i.e.,* the developer's effort for finding the answer).

3.3 Results

By analyzing the effectiveness results on our data set, we observed that none of the considered TR configurations consistently performs the best across all the queries. Figure 1 reports the percentage of queries for which each TR configuration achieved the best effectiveness in the entire data set. As it can be observed, the highest number of queries for which a TR configuration performed the best is no higher than 16% (*i.e.,* 162 out of 1,035 queries). This result is achieved by the default configuration of Lucene (E1C1). We also observed that the highest percentage of queries for which any of the TR configurations performed the best in a particular system varies between 12% and 43% (see Table 3). This means that if only one TR configuration were used for all queries of the 12 systems, between 57% and 88% of the queries would be suboptimally answered—and this is in the best case scenario, when the best performing TR configuration is chosen.

In addition, the difference in the results between choosing different TR configurations for the same query can be tangible, impacting the success (or failure) of the retrieval task. For example in `openjpa 2.0.1`, if we were to choose its best TR configuration E2C4 for all of its queries, we would come across cases like query 34, for which E2C4 retrieves the first relevant document on position 486, thus resulting in an unsuccessful search, as developers rarely look past the first 10 to 20 results. Choosing the configuration E5C1, on the other hand, would have returned the relevant document on the first position in the list of results for the same query. We also observed that the same TR configuration performs

Table 3: Percentage of queries for which the TR configurations obtain the the best effectiveness per system.

System	E1C1	E2C1	E2C2	E2C3	E2C4	E3C1	E3C2	E3C3	E4C1	E4C2	E4C3	E5C1	E6C1	E6C2	E6C3	E6C4	E6C5	E7C1	E7C2	E7C3	E7C4
apache-nutch-1.8	**43%**	0%	0%	10%	0%	3%	10%	0%	3%	0%	0%	7%	3%	7%	0%	10%	0%	3%	0%	0%	0%
apache-nutch-2.1	11%	2%	7%	**13%**	11%	9%	0%	2%	0%	2%	7%	2%	2%	9%	2%	2%	6%	11%	0%	2%	0%
bookkeeper-4.1.0	**15%**	0%	1%	4%	10%	11%	4%	1%	4%	4%	5%	5%	5%	7%	0%	6%	4%	10%	1%	2%	1%
commons-math3-3.0	**33%**	2%	6%	2%	2%	13%	4%	0%	4%	2%	2%	8%	10%	0%	0%	0%	0%	6%	2%	2%	0%
derby-10.9.1.0	14%	2%	1%	6%	**19%**	3%	1%	2%	4%	2%	8%	3%	11%	7%	1%	5%	3%	5%	2%	0%	0%
mahout-0.8	**26%**	5%	3%	3%	9%	6%	3%	3%	2%	0%	2%	5%	9%	17%	0%	5%	5%	0%	0%	0%	0%
openjpa-2.0.1	8%	3%	11%	14%	**26%**	5%	0%	0%	9%	2%	0%	8%	5%	2%	2%	0%	5%	3%	0%	2%	0%
pig-0.11.1	**12%**	0%	7%	10%	**12%**	4%	2%	1%	3%	0%	3%	10%	**12%**	4%	2%	6%	9%	0%	1%	0%	2%
pig-0.8.0	**11%**	2%	9%	9%	10%	4%	2%	1%	4%	2%	6%	16%	7%	5%	1%	6%	0%	4%	1%	2%	0%
solr-4.4.0	**13%**	0%	2%	7%	11%	6%	1%	0%	5%	2%	7%	11%	11%	8%	2%	9%	2%	1%	1%	2%	0%
tika-1.3	**23%**	3%	0%	3%	22%	3%	0%	3%	0%	0%	7%	5%	8%	7%	0%	7%	2%	3%	0%	3%	0%
zookeeper-3.4.5	**13%**	1%	7%	15%	7%	5%	1%	0%	4%	1%	9%	10%	5%	3%	3%	6%	3%	5%	1%	1%	0%
Average	*19%*	*2%*	*5%*	*8%*	*12%*	*6%*	*2%*	*1%*	*3%*	*1%*	*5%*	*7%*	*7%*	*6%*	*1%*	*5%*	*3%*	*4%*	*1%*	*1%*	*0%*

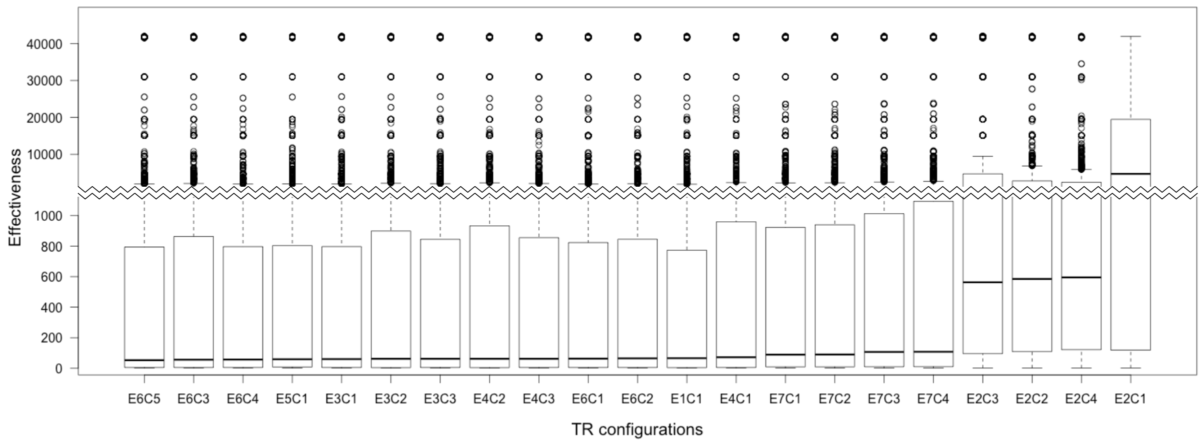

Figure 2: Effectiveness of the TR configurations in the whole data set sorted by median values (the lower the better).

quite differently across systems. For example, E2C4 is the configuration which leads to the best results over the largest set of queries in `derby-10.9.1.0` (*i.e.,* E2C4 has the best results among the 21 configurations for 19.2% of its queries), whereas the same configuration never obtains the best results among the 21 TR configurations for any of the queries in `apache-nutch-1.8`.

These examples are not outliers. Figure 2 shows the distribution of effectiveness (highest rank for the relevant method, so the lower the better) for all queries for each configuration. Data reveals that the effectiveness of the retrieval widely varies across configurations (as it can be seen by comparing the different boxes), and from query to query (as it can be seen from the large boxes, *i.e.,* interquartile differences, and from the presence of several outliers exhibiting a rank having value of orders of magnitudes greater than the median). For example, E2C4 and E2C3 (corresponding to LDA) are the 2nd (12%) and 3rd (9%) best configurations based on the number of queries for which they obtain the best effectiveness (see Figure 1). If someone is determined to use LDA for their task, she might be tempted to choose one of these two configurations. However, based on median effectiveness (see Figure 2), these two configurations are ranked as 18th (median effectiveness 563) and 20th (median effectiveness 585) among the 21 configurations, respectively. In other words, their performance over all the systems and queries is nearly the worst.

The complete results of this exploratory study can be found in our replication package. These results underline two facts: (i) choosing one TR configuration for all queries in a system or across several systems leads to suboptimal results, and (ii) in some cases, the difference between the optimal results that could be achieved using the best TR configuration for each individual query and those obtained using only one TR configuration for all queries is significant. As shown above, *choosing the right TR configuration can make the difference between a successful and an unsuccessful search.* Since developers get easily discouraged using a particular tool if the results retrieved are not good even for a few instances, choosing the TR configuration that leads to the best results for each query could have a significant impact on the adoption of TR approaches by practitioners. In the next section we describe our approach, which premiers query-level TR configuration on software corpora.

4. AN APPROACH FOR QUERY-BASED TR CONFIGURATION: QUEST

As any technique based on supervised learning, QUEST follows two major steps. The first one is a training step, where a classifier is built from a training data set. The second step is recommending the TR configuration that performs the best for a given query, based on the trained classifier. The idea behind QUEST is that the performance of TR configurations varies according to different properties of the queries. Since both the training and recommendation steps make use of such properties, we provide first a brief overview of them.

4.1 Query Properties and Measures

Previous work on query quality prediction [25] and query reformulation [24] in software engineering adopted a series of Information Retrieval measures that capture linguistic and statistical properties of queries and a software corpus. These measures were used to predict whether a query will lead to relevant or irrelevant TR results [25], given a TR configuration, and to automatically find the reformulation for a query that leads to the best results, given a TR configuration [24]. In the context of this paper, we use these measures as indicators of the performance of particular TR configurations, since they capture properties of the query, software corpus, and the results retrieved by a TR approach.

Existing query and corpus measures are categorized into pre-retrieval and post-retrieval [12], depending on the moment when they are computed and the type of information they capture.

Pre-retrieval properties and measures. Pre-retrieval measures are computed before the query is run, and measure linguistic and statistical properties of the query and its relationship with the software corpus. Pre-retrieval measures assess properties such as, coherency, specificity, similarity, and term relatedness [12].

Coherency indicates how focused a query is on a particular topic [28, 56]. The coherency of a query is usually measured as the level of inter-similarity between the documents in the corpus containing the query terms. The more similar the documents are, the more coherent the query is.

Specificity refers to the ability of the query to represent the current information need and discriminate it from others [27, 41]. The main idea behind this property is that a query composed of terms commonly used in the corpus is considered having low specificity, as it is hard to differentiate the relevant documents from non-relevant ones based on its terms. For example, when searching source code, the query "initialize members" could have low specificity, if a comment containing this text would be found in most class constructors in a system. Specificity measures are usually based on the query terms' distribution over the collection of documents, but the way this information is captured differs from measure to measure.

The *similarity* between the query and the entire document collection is another property that reflects an aspect of the query and corpus [51]. The argument behind this type of measure is that it is easier to retrieve relevant documents for a query that is similar to the corpus, since high similarity potentially indicates the existence of many relevant documents in the corpus to retrieve from.

Finally, *term relatedness* measures make use of term co-occurrence statistics in order to assess the performance of a query [26]. The terms in a query are assumed to be related to the same topic and are, thus, expected to occur together frequently in the corpus.

Post-retrieval properties and measures. While pre-retrieval measures capture some general characteristics of the query and corpus, they do not take into consideration the list of results returned by a TR engine. Post-retrieval measures rely on the analysis of the search results, that is, the list of documents in the corpus ranked highest in response to the query.

Robustness-based measures evaluate how robust the results are to perturbations in the query and the documents in the result list [53, 55]. Some of these measures assess the robustness of the result list to small modifications of the query. When small changes in the query cause large changes in the search results, the confidence in the capacity of the query to capture the essential information diminishes. Document perturbation measures, on the other hand, rely on injecting the top documents in the result list with noise and re-ranking them, measuring the difference in their ranks before and after the perturbation. For a robust query, small perturbations of the documents in the result list should not result in significant changes in their ranking.

Score distribution-based methods analyze the similarity between the query and the results, which are used to rank the results of the retrieval [47, 56]. For example, the highest retrieval score (i.e., similarity) and the mean of top scores indicate query performance, since, in general, low scores of the top-ranked documents indicate some difficulty in retrieval.

Clarity-based methods directly measure the "focus" (clarity) of the search results with respect to the corpus. Due to the considerable execution time of the clarity-based measures, we did not include them in the current implementation of our approach.

4.2 Training Phase

The starting point for building QUEST's classifier is a *training set* built from a collection of queries, their properties, and their respective relevant documents. QUEST is a general approach, which can be applied in the context of any SE task making use of TR techniques. Therefore, what each query in the collection represents and how it is collected can vary according to the SE task at hand. In the case of feature and bug localization, on which we focus in this paper, the queries and relevant documents represent, respectively, text extracted from change requests and the methods modified in response to the requests (as in the case of our exploratory study in Section 3).

The next are the steps followed by QUEST during the training phase:

1. For each query in the data collection, QUEST computes the values of a predefined list of n query property measures. The current version of our approach uses 21 pre-retrieval and seven post-retrieval query measures shown to be useful in other query-related tasks in SE [25, 24]. The 28 measures currently implemented in QUEST are listed in Table 4 and fully described in our replication package. As post-retrieval measures require the analysis of retrieval results for each query, we use the results provided by Lucene to compute such measures. We chose Lucene, as the exploratory study indicated that its standard configuration is one of the better ones among the ones we studied (*i.e.,* E1C1). Using other TR engine or configuration for computing the post-retrieval measures does not change our approach. Future work will investigate the use of other TR configurations for the computation of the post-retrieval measures.
2. For each query, QUEST then applies, one by one, a set of TR configurations to rank the documents relevant to the query and identifies the one performing the best for each query. In feature and bug localization the best configuration is the one that retrieves any of the relevant artifacts to a query on the highest position in the list of results (*i.e.,* lowest effectiveness values).

Table 4: Query property measures implemented by QUEST.

Property	Measure
Specificity (PRE)	Maximum of the Inverse Document Frequency
	Standard deviation of the Inverse Document Frequency
	Average Inverse Collection Term Frequency
	Maximum Inverse Collection Term Frequency
	Standard deviation of the Inverse Collection Term Frequency
	Average entropy
	Median entropy
	Maximum entropy
	Standard deviation of the entropy.
	Query Scope
	Simplified Clarity Score
Coherency (PRE)	Average of the Variances
	Maximum of Variances
	Sum of Variances
	Coherence Score
Similarity (PRE)	Average of the collection-query similarity
	Maximum of the collection-query similarity
	Sum of the collection-query similarity
Term Relatedness (PRE)	Average Pointwise Mutual Information
	Maximum Pointwise Mutual Information
Robustness (POST)	Subquery Overlap
	Robustness Score
	First Rank Change
	Clustering Tendency
	Spatial Autocorrelation
Score distribution (POST)	Weighted Information Gain
	Normalized Query Commitment

3. Then, each data point in the training set represents a query described by a vector of n attributes corresponding to the values of the query properties and one attribute corresponding to the best TR configuration for the query. This latter represents the class label of a data point.
4. Finally, a supervised learning algorithm is run on the training set. Currently, QUEST uses classification trees [10], which produce human-understandable rules and implicitly perform feature selection. This is important for QUEST, as it reduces the sensitivity to the choice of query property measures. In other words, given as input all n measures of a query, the classification tree will automatically determine those properties that are relevant for the classification, with little overhead. Classification trees are generally suitable to solve problems where the goal is to determine the values of a categorical variable based on one or more continuous and/or categorical variables. In our approach, the categorical dependent variable is represented by the best TR configuration to use for a particular query, while the independent variables are the 28 query property measures. The classifier uses the training data to automatically select the independent variables and their interactions that are most important in determining the dependent variable to be explained. Future work will investigate other learning algorithms.

Two approaches can be followed to train the classifier: *within-project* (*i.e.,* training a classifier for each software system independently) and *cross-project* (*i.e.,* training a global classifier for all the considered systems). Cross-project training can be useful when training data for a specific software system may not be obtained. Within-project training has been shown to outperform cross-project training in previous work [24] and our experimental evaluation from Section 5 confirms this trend also for QUEST.

A classification tree is the outcome of QUEST's training stage. In this tree, each root-to-leave path represents a series of conditions or rules on the query properties that need to be satisfied in order to reach a class label, given by the path's leaf.

4.3 Recommendation Phase

For a new query, QUEST computes the same n query properties considered in the training phase. Based on these values, QUEST uses the classification tree obtained in the training phase to automatically determine the TR configuration that is expected to perform the best for the query. Finally, the selected TR configuration is run to retrieve the documents relevant to the query.

5. QUEST EVALUATION STUDY

We conducted an empirical study to assess QUEST's prediction accuracy and to compare it with the traditional way of applying TR approaches in software engineering (*i.e.,* one TR configuration for an entire data set), in the context of feature and bug localization.

The study aims at answering the following two research questions:

RQ$_1$. *How accurately does* QUEST *predict the best TR configuration for a give query?*

RQ$_2$. *Does* QUEST *improve feature and bug localization compared to using a single TR configuration for a software system?*

5.1 Study Design

We built an instance of QUEST to perform the empirical study. The methodology followed in this study is in many ways similar to the exploratory study presented in Section 3, so we will not go into details whenever the same data or settings were used and previously explained.

Data set. We used the same data set as in our exploratory study, consisting of 1,035 queries extracted from resolved issues (*i.e.,* feature requests and bug reports) of 12 versions of ten software systems, described in Table 1. As a reminder, each query in the data set was extracted from either the title, the description or the concatenated title and description of an issue. For each query, the set of methods that were modified in response to its corresponding change request are known and represent the gold set used to determine the performance of the TR configurations and QUEST. The text corpus of a system contains its methods as documents and each document contains the identifiers and comments extracted from a method. Both the queries and the corpus of each system were processed following the same procedure as in the exploratory study.

Performance measurement. We measure the performance of a TR configuration and of QUEST as its effectiveness, *i.e.,* the position of the first relevant document in the ranked list of retrieved results. As mentioned in Section 3 this is an inverse measure, so a lower value is better (*i.e.,* an effectiveness value of 1 is the best possible result, which means that one of the relevant methods is ranked the highest in the list of retrieved documents).

TR configurations. It is impractical to build classification trees that select among 21 categories, so we limited the number of TR configurations used in this study to eight. We selected the ones with the highest percentage of queries for which they obtain the best effectiveness from our exploratory study (Table 2): E1C1, E2C3, E2C4, E3C1, E4C3, E5C1, E6C1, and E6C2. The rationale behind this choice is to ensure that there are enough samples in each category. QUEST will consider each of these as a category and will recommend applying one of them for a given query, based on its properties.

Training strategy. We used both within- and cross-project strategies to train QUEST. In the *within-project* case, one classification model was individually trained on each system and a stratified 4-fold cross-validation was performed following the next steps:

1. randomly divide the set of queries of a system into four equal subsets (if possible), containing approximately the same percentage of samples of each target class as the complete set;
2. set aside one of the subsets as a test set and build the classification model with the queries in the remaining subsets (*i.e.,* the training set);
3. use the classification model built on the training set to identify the best TR configuration for the queries in the test set;
4. repeat this process, setting aside each subset in turn.

Note that each query is used only once in the test set. In the case of the *cross-project* training, the queries of 11 of the systems in the data set are used for training and the queries from the remaining project are used for evaluation. This is repeated such that the queries in each project are tested.

In order to answer $\mathbf{RQ}_1$, we evaluate the fitness of the training strategies by computing their top-k recommendation accuracy, this is, for each query, if the TR configuration recommended by QUEST is in the top-k TR configurations (*i.e.,* the best k configurations according to the performance measurement), we consider it as a success; otherwise, we consider it as a failure.

Baselines. In order to answer $\mathbf{RQ}_2$, we considered each of the eight TR configurations selected for the evaluation as baselines and ran them for the entire data set. Then we compared the results of applying each individual TR configuration to each system with the results obtained by running the TR configuration recommended by QUEST for each individual query. The comparison was based on the effectiveness of the approaches and was made by considering the number of queries for which the effectiveness measure obtained by QUEST improves, preserves, or deteriorates each of the baselines' effectiveness.

Table 5: QUEST's top-k recommendation accuracy when using within- and cross-project training.

Training strategy	System	Top 1	Top 2	Top 3
Within-project	apache-nutch-1.8	57%	75%	86%
	apache-nutch-2.1	39%	62%	69%
	bookkeeper-4.1.0	23%	58%	69%
	commons-math3 3.0	62%	75%	79%
	derby-10.9.1.0	31%	53%	73%
	mahout-0.8	53%	70%	78%
	openjpa-2.0.1	39%	70%	73%
	pig-0.8.0	20%	34%	48%
	pig-0.11.1	29%	45%	58%
	solr-4.4.0	22%	39%	56%
	tika-1.3	53%	63%	65%
	zookeeper-3.4.5	30%	45%	70%
	Average	***38%***	***57%***	***69%***
Cross-project	Average	13%	24%	35%

The sets of results were also analyzed through statistical analysis using the Mann-Whitney test [14]. We chose this test as we cannot assume normality of data and the test does not make normality assumptions. The results are interpreted as statistically significant at $\alpha < 0.05$. However, since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [29]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on.

5.2 Results

We present and discuss the results for each research question separately.

5.2.1 $\mathbf{RQ}_1$—*Classification Accuracy*

Table 5 reports the top-k recommendation accuracy achieved by QUEST when using within- and cross-project strategies to train the classifier. The data reveals that the classification model based on cross-project training is significantly less accurate than the models built for each individual project. As expected, the accuracy of both the training within- and cross-project strategies increases as the top set increases its size k. The accuracy of the within-project trained classification models varies from system to system (see Table 5). The best top-1 recommendation accuracy is achieved for the `commons-math3 3.0` project, in which the best possible TR configuration was selected 62% of the times. This percentage increases to 79% when considering the top-3 TR configurations as success factor. The worst top-1 accuracy corresponds to `pig 0.8.0`, where the classification model predicted the best TR configuration in only 20% of the cases. Recommending any of the top-3 configurations increases the accuracy to 48% for this system. On average, the within-project training ensures a success in the selection of the top-k configuration in 38% of cases when $k = 1$, 57% when $k = 2$, and 69% when $k = 3$ (see Table 5).

5.2.2 $\mathbf{RQ}_2$—*Comparison with Baselines*

In order to determine if query-based configuration brings improvement over the traditional way of applying TR-based approaches for feature and bug localization, we compared QUEST with the eight considered baselines, according to the procedure described in Section 5.1. Given the lower accuracy

Table 6: Percentage of queries for which QUEST improved (in parenthesis, median of improved positions), preserved and deteriorated (in parenthesis, median of deteriorated positions) the retrieval performance with respect to each baseline.

Baseline	Improved (med. improv.)	Preserved	Deteriorated (med. deter.)
E1C1	27.8% (-84)	53.1%	19.1% (233.5)
E2C3	71.9% (-412)	16.6%	11.5% (159)
E2C4	67.8% (-367)	18.2%	14.0% (763)
E3C1	44.6% (-40)	25.5%	29.9% (66)
E4C3	41.3% (-53)	30.5%	28.2% (99)
E5C1	51.5% (-42.5)	21.4%	27.1% (75)
E6C1	35.7% (-48)	31.0%	33.3% (61)
E6C2	41.8% (-60.5)	29.0%	29.2% (66)
Average	*47.8% (-138.4)*	*28.2%*	*24.0% (190.3)*

results of the cross-project training described in the previous subsection, we focus here only on the recommendations using the within-project training model.

Table 6 shows how QUEST's recommended configuration compares with the baselines, aggregated for all queries (*i.e.,* 1,035). The rows of the table correspond to each of the eight baselines, and the columns show the percentages of queries from the data set for which QUEST leads to an improved effectiveness measure (*i.e.,* the TR configuration recommended by QUEST led to a better effectiveness measure than the TR baseline on that row), preserved the same effectiveness results (*i.e.,* the TR configuration recommended by QUEST returned the same effectiveness value as the TR baseline on the row), or deteriorated the effectiveness measure (*i.e.,* the TR recommended by QUEST returned a higher effectiveness value than the TR baseline), respectively. The values in parenthesis indicate the median change in the effectiveness.

It must be pointed out that a lower effectiveness value indicates a better result, *i.e.,* a lower rank in the list of retrieved results for the first relevant document to the query. "Preserved" means that the effectiveness measure was the same between the baseline and the TR configuration recommended by QUEST; however, in some cases this effectiveness actually represents the best value among all TR configurations considered. Therefore, in some "preserved" cases, QUEST's recommendation led to the best results, even if the baseline obtained the same value. When a TR-based feature or bug localization technique improves in some cases compared to a baseline and it deteriorates in other cases, it is important to preserve the best results (*i.e.,* where improvements cannot be achieved). We consider these cases also a successful recommendation; Table 7 lists these cases for the comparison of QUEST with each baseline.

The results indicate that QUEST improves the effectiveness for 47.8% of the queries, maintains the effectiveness for 28.2% of queries, and deteriorates the effectiveness in 24.0% of the cases, when averages across all eight baselines are considered. Therefore, using the TR configuration recommended by QUEST for each query results in an improved or preserved effectiveness for 76% of the queries. This means that there are twice as many queries with improved results than those for which the results deteriorate, compared to the average baseline results. We note that among the cases where the effectiveness is preserved, QUEST's TR configuration led to the best effectiveness results in more than half (56.8%) of the cases (see Table 7).

Table 7: Queries for which the results were preserved by QUEST, but represent optimal results.

Baseline	# Preserved	# Preserved but Best	% Preserved but Best
E1C1	406	175	43.1%
E2C3	127	55	43.3%
E2C4	139	84	60.4%
E3C1	195	127	65.1%
E4C3	233	137	58.8%
E5C1	164	107	65.2%
E6C1	237	147	62.0%
E6C2	222	146	65.8%
All	*1,723*	*978*	*56.8%*

When compared to individual TR baselines, QUEST always leads to more cases of improvement than deterioration of the effectiveness. The percentage of improved queries ranges from 27.8%, when compared to E1C1 (with only 19.1% of deterioration), to 71.9%, when compared to E2C3 (with only 11.5% of deterioration). We note, however, that for E1C1 43.1% of the preserved results represent the best results among all TR configurations and therefore are successful recommendations, as the effectiveness cannot be improved. Also, in E6C1, which has the second lowest percentage of improved queries, 62% of the preserved effectiveness results are optimal. The highest improvement in effectiveness was obtained over E2C3. 71.9% of the results were improved, and 43.3% of the preserved results are optimal. One must also note that the median size of improvement in the effectiveness is somewhat lower than the size of the deterioration (138 vs. 190—see Table 6). However, considering the larger number of improvements than deteriorations, we consider the results promising.

We performed also the Mann-Whitney statistical test between the results obtained by the recommendation of QUEST and those obtained by each of the baselines, in order to see if the differences between them are statistically significant. After Holm's correction, the results revealed three baselines for which there is no statistically significant difference between the results of QUEST and that of the baseline. These baselines are E2C3 (adjusted p-value of 3.95E-53), E2C4 (adjusted p-value of 7.44E-31), and E6C1 (adjusted p-value of 3.89E-03). While the results of the Mann-Whitney test did not show statistically significant difference between the performance of QUEST and the rest of the TR baselines, the results do show that overall QUEST improves the effectiveness measure.

As previously said, there is no single TR configuration that performs the best across all systems (see Table 3 in Section 3). Therefore, we performed a more in-depth analysis of the comparison between the results of the baseline performing the best in each system and QUEST. Table 8 presents the per-system results of this comparison, in terms of the percentage of queries for which the results were improved, preserved, or deteriorated. We observe that QUEST leads to more improved queries than deteriorated ones for all but two systems: `solr-4.4.0` and `zookeeper-3.4.5`, where the percentage of improved queries lags behind the percentage of deteriorated ones. Incidentally, these are two of the systems where the predictor had the lowest accuracy (see Table 5). However, from the cases where the effectiveness was preserved, 50% in `solr-4.4.0` and 38.3% in `zookeeper-3.4.5` represented optimal values that were preserved by QUEST's recommendation, when compared to the respective best TR

Table 8: Percentage of queries per system for which QUEST improved (in parenthesis, median of improved positions), preserved and deteriorated (in parenthesis, median of deteriorated positions) the retrieval performance with respect to the best TR baseline.

System	Best baseline	Improved (med. improv.)	Preserved	Deteriorated (med. deter.)
apache-nutch-1.8	E1C1	29.2% (-3)	66.7%	4.2% (1)
apache-nutch-2.1	E2C3	61.5% (-111)	28.2%	10.3% (34)
bookkeeper-4.1.0	E1C1	30.6% (-5)	53.2%	16.1% (7.5)
commons-math3-3.0	E1C1	19.4% (-2)	69.4%	11.1% (23.5)
derby-10.9.1.0	E2C4	56.0% (-1,062)	34.7%	9.3% (770)
mahout-0.8	E1C1	25.0% (-20)	72.7%	2.3% (9,457)
openjpa-2.0.1	E2C4	37.8% (-1,948)	48.9%	13.3% (420)
pig-0.11.1	E1C1	33.8% (-312)	43.8%	22.5% (551.5)
pig-0.8.0	E5C1	39.7% (-65)	28.6%	31.7% (531)
solr-4.4.0	E6C1	24.1% (-10.5)	36.1%	39.8% (25)
tika-1.3	E1C1	37.8% (-93)	37.8%	24.4% (88)
zookeeper-3.4.5	E1C1	10.4% (-18)	76.4%	13.2% (68)
Average		*33.8% (-304.1)*	*49.7%*	*16.5% (1,000.3)*

baselines (*i.e.,* E6C1 and E1C1). Over the entire data set, the average number of queries with worse results than the best individual baseline (*i.e.,* 16.5%) is lower than the average of improved queries across all systems (*i.e.,* 33.8%), showing the potential of using TR configurations at query level over system level.

We answer $\mathbf{RQ}_2$ as follows. QUEST's recommended TR configurations lead to better results for 47.8% of the queries, it preserves the effectiveness for 28.2% of the queries, and it deteriorates the results for 24% of the queries. The results are statistically significant for three of the eight baselines. While the results indicate that there is room for improvement, they are promising and support the use TR configurations at query level.

6. THREATS TO VALIDITY

Regarding the *construct validity*, both the exploratory study (Section 3) and the empirical study (Section 5) rely on a query performance measure (*i.e.,* the effectiveness) that is widely used in TR-based feature and bug localization studies and provides a proxy measure of the developer's effort in a feature location task.

Several co-factors can influence our results (*internal validity*). We automatically extracted the set of queries used in both studies from the online issues trackers of the object systems. Such queries are approximations of actual user queries and we believe that they resemble real usage scenarios. Nonetheless, increasing the size of the data set could affect the classification model and its accuracy. Another influencing co-factor is the set of query property measures implemented by QUEST. We selected measures that capture different aspects of the queries and have shown to be useful in other SE tasks (*e.g.,* query quality prediction and query reformulation). The eight TR configurations selected for the empirical study were the ones with most best ranked instances, which ensures better distribution of classes in the training data. Using a different set of TR configurations can affect the accuracy of the predictors. We believe, however, that QUEST will improve or preserve the performance of traditional TR-based approaches in most of the cases. The last co-factor is the computation of the post-retrieval query properties based on Lucene. The results can vary when using another TR engine for this task.

Threats to *conclusion validity* concern the relationship between treatment and outcome. Where appropriate, we used non-parametric statistical tests (Mann-Whitney) to show statistical significance for the obtained results.

Concerning the generalization of the obtained results (*external validity*), we selected queries from 12 versions of ten Java software systems from different domains and with different size. A larger set of queries from different systems and written in other programming languages would clearly strengthen the results from this perspective. Moreover, we used eight TR configurations when training the classification models. The results might vary when using other TR configurations. Finally, we only evaluated the proposed approach in the context of feature and bug localization. Thus, we cannot (and do not) generalize the results to other SE tasks.

7. CONCLUSIONS AND FUTURE WORK

We proposed a novel approach, QUEST, which is the first to perform query-specific TR configuration selection in the context of a software engineering task. QUEST uses machine learning techniques in order to learn the best TR configuration to be used for each query. The learning is based on a set of linguistic and statistical properties of the query itself, the software corpus, and the list of results returned for it. In an exploratory study, we found that choosing any individual TR configuration to use for all the queries formulated for a system or for an entire data set leads to suboptimal results. Then, in our empirical evaluation, we showed that QUEST is able to correctly assign one of the top-3 performing TR configurations for a query with 69% accuracy on average and, overall, it leads to improved results compared to any individual TR configuration applied to the entire data set.

While we consider the results positive and promising, there is room for improvement. For a few systems, and when compared with a few TR configurations, QUEST lead to slightly more queries for which the results deteriorated than the number for which it improved. Our future work will focus on determining what properties of the software or queries led to this suboptimal performance and will focus on overcoming these cases. Other directions of future work include: (i) considering more TR configurations in order to determine a more variate set of post-retrieval query measures, (ii) considering different classifiers, (iii) including more TR configurations to choose from in QUEST, and (iv) experimenting with the size of the training set to see the impact it has on the accuracy of the prediction. Last but not least, we will evaluate QUEST in the context of other software engineering tasks where TR is used.

We believe the work presented in this paper opens the door to a new research direction in TR-based software engineering, which will take the query into the consideration when applying TR techniques to SE tasks. Since developers often get discouraged from using tools if the results they retrieve are not correct even for a few instances, we believe that devising and perfecting an approach like QUEST could have a positive impact on the adoption of TR approaches by practitioners.

8. ACKNOWLEDGMENTS

Laura Moreno and Andrian Marcus are supported in part by the National Science Foundation grants CCF-1526118 and CCF-085706. Sonia Haiduc is supported in part by the National Science Foundation grant CCF-1526929.

9. REFERENCES

[1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *Proceedings of 16^th IEEE Int'l Conf. on Program Comprehension*, pages 103–112, Amsterdam, The Netherlands, 2008. IEEE CS Press.

[2] Amati, G Rijsbergen, Van Rijsbergen, and C. J. Probabilistic models of information retrieval based on measuring the divergence from randomness. *ACM Transactions on Information Systems*, 20(4):357–389, 2002.

[3] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of 16^th IEEE Int'l Conf. on Software Maintenance*, pages 40–51, San Jose, California, USA, 2000. IEEE CS Press.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[5] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *Proceedings of 28^th IEEE/ACM Int'l Conf. on Software Engineering*, pages 361–370, Shanghai, China, 2006. IEEE CS Press.

[6] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft. Configuring latent Dirichlet allocation based feature location. *Empirical Software Engineering*, 19(3):465–500, Aug. 2012.

[7] D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt. Understanding LDA in source code analysis. In *Proceedings of the 22Nd Int'l Conf. on Program Comprehension*, pages 26–36, New York, NY, USA, 2014. ACM.

[8] D. Binkley and D. Lawrie. Learning to rank improves IR in SE. In *Proceedings of the 20th IEEE Int'l Conf. on Software Maintenance and Evolution*, pages 441–445, Sept. 2014.

[9] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[10] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and Regression Trees*. Chapman and Hall, 1984.

[11] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 Int'l Workshop on Mining Software Repositories*, pages 105–111, New York, NY, USA, 2006. ACM.

[12] D. Carmel and E. Yom-Tov. Estimating the query difficulty for information retrieval. In *Proceedings of the 33rd Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 911–911, New York, NY, USA, 2010. ACM.

[13] S. Clinchant and E. Gaussier. Information-based models for ad hoc IR. In *Proceedings of the 33rd Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 234–241, New York, NY, USA, 2010. ACM.

[14] W. J. Conover. *Practical Nonparametric Statistics.* Wiley, 3rd edition edition, 1998.

[15] A. De Lucia, M. Risi, G. Tortora, and G. Scanniello. Clustering algorithms and latent semantic indexing to identify similar pages in web applications. In *9th IEEE Int'l Workshop on Web Site Evolution*, pages 65–72, Oct. 2007.

[16] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[17] D. Falessi, G. Cantone, and G. Canfora. Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. *IEEE Transactions on Software Engineering*, 39(1):18–44, Jan. 2013.

[18] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *19th Int'l Conf. on Software Maintenance*, pages 23–32, Amsterdam, The Netherlands, 2003. IEEE Computer Society.

[19] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *Proceedings of the 15th IEEE Int'l Conf. on Software Maintenance*, pages 351–360, Sept. 2009.

[20] X. Geng, T.-Y. Liu, T. Qin, A. Arnold, H. Li, and H.-Y. Shum. Query dependent ranking using k-nearest neighbor. In *Proceedings of the 31st Annual Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 115–122, New York, NY, USA, 2008. ACM.

[21] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. On integrating orthogonal information retrieval methods to improve traceability recovery. In *Proceedings of the Int'l Conf. of Software Maintenance*, pages 133–142, 2011.

[22] S. Grant and J. Cordy. Estimating the optimal number of latent concepts in source code analysis. In *10th IEEE Working Conf. on Source Code Analysis and Manipulation*, pages 65–74, Sept. 2010.

[23] S. Grant, J. R. Cordy, and D. B. Skillicorn. Using heuristics to estimate an appropriate number of latent topics in source code analysis. *Science of Computer Programming*, 78(9):1663–1678, 2013.

[24] 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 Int'l Conf. on Software Engineering*, pages 842–851, Piscataway, NJ, USA, 2013. IEEE Press.

[25] S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus. Automatic query performance assessment during the retrieval of software artifacts. In *Proceedings of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering*, pages 90–99, New York, NY, USA, 2012. ACM.

[26] C. Hauff. Predicting the effectiveness of queries and retrieval systems. *SIGIR Forum*, 44(1):88–88, Aug. 2010.

[27] B. He and I. Ounis. Inferring query performance using pre-retrieval predictors. In *SPIRE*, pages 43–54, 2004.

[28] J. He, M. Larson, and M. De Rijke. Using coherence-based measures to predict query difficulty. In *Proceedings of the IR Research, 30th European Conf. on Advances in Information Retrieval*, pages

689–694, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, 6:65–70, 1979.

[30] I.-H. Kang and G. Kim. Query type classification for web document retrieval. In *Proceedings of the 26th Annual Int'l ACM SIGIR Conf. on Research and Development in Informaion Retrieval*, pages 64–71, New York, NY, USA, 2003. ACM.

[31] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[32] T.-Y. Liu. *Learning to Rank for Information Retrieval*. Springer Berlin Heidelberg, 2011.

[33] T.-Y. Liu. Query-dependent ranking. In *Learning to Rank for Information Retrieval*, pages 113–121. Springer Berlin Heidelberg, 2011.

[34] K. K. Lo, M. K. Chan, and E. Baniassad. Isolating and relating concerns in requirements using latent semantic analysis. In *Proceedings of ACM SIGPLAN Int'l Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 383–396, Portland, Oregon, USA, 2006. ACM Press.

[35] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 378–388, New York, NY, USA, 2013. ACM.

[36] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 52(9):972–990, Sept. 2010.

[37] A. Marcus and G. Antoniol. On the use of text retrieval techniques in software engineering. In *34th IEEE/ACM Int'l Conf. on Software Engineering, Technical Briefing*, 2012.

[38] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of 11th Working Conf. on Reverse Engineering*, pages 214–223, Delft, The Netherlands, 2004. IEEE CS Press.

[39] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM Int'l Conf. on Automated Software Engineering*, pages 263–272, Nov. 2011.

[40] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*, pages 522–531, Piscataway, NJ, USA, 2013. IEEE Press.

[41] V. Plachouras, B. He, and I. Ounis. University of Glasgow at trec 2004: Experiments in web, robust, and terabyte tracks with terrier. In *Proceedings of the 13th Text REtrieval Conf.* NIST Special Publication, 2004.

[42] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st Annual Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 275–281, New York, NY, USA, 1998. ACM.

[43] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[44] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conf. on Mining Software Repositories*, pages 43–52, New York, NY, USA, 2011. ACM.

[45] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. pages 109–126, 1996.

[46] Z. Shi, J. Keung, and Q. Song. An empirical study of bm25 and bm25f based feature location techniques. In *Proceedings of the Int'l Workshop on Innovative Software Development Methodologies*, pages 106–114, New York, NY, USA, 2014. ACM.

[47] A. Shtok, O. Kurland, and D. Carmel. Predicting query performance by querydrift estimation. In *2nd Int'l Conf. on Theory of Information Retrieval*, 2009.

[48] S. Thomas, M. Nagappan, D. Blostein, and A. Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 39(10):1427–1443, Oct. 2013.

[49] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *Proceeding of the 20th IEEE Int'l Conf. on Software Maintenance and Evolution*, pages 171–180, Sept. 2014.

[50] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 181–190, Sept. 2014.

[51] J. Yang and L. Tan. Inferring semantically related words from software context. In *9th IEEE Working Conf. on Mining Software Repositories*, pages 161–170, 2012.

[52] 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 Int'l Symposium on Foundations of Software Engineering*, pages 689–699, New York, NY, USA, 2014. ACM.

[53] E. Yom-Tov, S. Fine, D. Carmel, and A. Darlow. Metasearch and federation using query difficulty prediction. In *28th Annual Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval, Workshop on Query prediction and its applications*, 2005.

[54] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, Apr. 2004.

[55] Y. Zhou and W. B. Croft. Ranking robustness: a novel framework to predict query performance. In *15th ACM Int'l Conf. on Information and Knowledge Management*, 2006.

[56] Y. Zhou and W. B. Croft. Query performance prediction in web search environments. In *30th Annual Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 543–550. ACM, 2007.