On the Influence of Latent Semantic Analysis Parameterization for Bug Localization

Marcelo de Almeida Maia [1]
Allysson Costa e Silva [1]
Ilmério Reis da Silva [1]

Abstract: The bug localization problem has benefited from modern information retrieval techniques, such as Latent Semantic Analysis. There are many factors that influence the quality of results of this approach, such as, stop-words, term-document matrix transformations, dimensionality reduction and filtering criteria of the corpus. In this paper, we study the effect of different combinations for these factors on the impact of the accuracy of the query results in the proposed technique for bug localization. Bugs of three real-world software systems were analyzed with different combinations of input parameters for the LSA technique. Our results suggest that the term-document matrix transformations and filtering criteria of the corpus have major influence in the quality of the result and that the combination of adequate individual parameter values does not necessarily produce the best combination. Furthermore, some general guidance for parameterization of the LSA technique for bug localization could also be suggested from the observed results.

1 Introduction

One important problem in program comprehension is the localization of bugs in source code. A starting point for this task is generally the report of the bug. An assumption to design a helpful approach to find the bug in the source code is that the terms used in the report of the bug may have some similarity degree with the terms of the source code fragments related to the bug, because the scope of bug searching could be narrowed or driven by a ranked list of code elements resulted from a query using bug terms in the indexed space of source code terms. The advance of techniques for Information Retrieval - IR has opened opportunities to this problem [1, 2, 3].

Even though important results were achieved during the last decade, there are some external factors that may hinder the application of IR techniques for bug localization. For instance, if terms used in the source code are extremely codified, then it is hard to expect that natural language terms of the user would match those coded terms. But even when terms used in source code are adherent to user's terms, internal factors of the information retrieval technique are also expected to influence on the result of localization. Indeed, De Lucia et

[1]Faculdade de Computação, Universidade Federal de Uberlândia, CEP 38400-902
{marcmaia,ilmerio@facom.ufu.br}, allcostaes@yahoo.com.br

al. [4] found evidences that the IR engine is an influence factor on the retrieval accuracy. The effective use of information retrieval techniques still seems to depend on tuning internal components of the IR engine, but also on tuning input parameters for the engine.

This paper focuses on the study of different combination of input parameters for an information retrieval technique and their influence on the accuracy of the query results. Latent semantic indexing has been extensively studied as a possible alternative for feature location and was chosen as the target IR technique for this study [3, 5, 6].

There are several parameters that can influence the results of an IR technique in general contexts, such as the dimensionality reduction, or even in the software maintenance context [7, 8]. This work will focus on the influence of input parameters of LSI techniques to the accuracy and performance of bug localization activity.

Our hypothesis for this study is that weighting functions for term-document matrix and filtering methods of the corpus can have an important influence on the performance of the LSI approach to help bug localization.

The rationale for the hypothesis of weighting functions is based on the fact that different types of terms do not have the same level of importance in the source code. Are the names of methods as important as the name of local variables when describing the semantics of a source code "document"? Can be the same asked for the class names? One goal of this paper is to evaluate if different weighting functions for terms extracted from method and class names have impact on the result of the performance of the LSI method.

The rationale for the hypothesis of corpus filtering is that, especially for bug localization, it is not necessary to search for the bug in the whole source code. There are some possible filters that may reduce the search space and this reduction may provide better search results. In this paper, we will evaluate two possibilities for filtering the source code, and assess the impact of this filtering on the execution time of the approach and on the precision and recall of the LSI results. This hypothesis would be possibly rejected because when the source code is filtered, the code with bug may be removed from the search space having a negative impact in recall.

This paper is organized as follows. In Section II, the study setting is presented including some background about the whole process. In Sections III and IV, we present the results and discussion of the experimental study, respectively. In Section V, discuss the threats to validity and finally in Section VI and VII, related work and conclusions are presented.

2 The Study Setting

This section describes the proposed bug localization process, detailing the proposed experimental setting.

2.1 The Bug Localization Process

The bug localization process using an IR technique resembles the process of finding information in the Web using a search engine. The chosen IR engine for this study is LSI [9]. Although there is no fundamental reason for preferring LSI to other IR methods, since other approaches would be also be considered, LSI is a reasonably well-studied technique for traceability recovery [10, 11, 4] and the results have shown that it addresses problems of polysemy and synonymy and also it has worked better than vector space models and Bayesian classifiers [5].

Figure 1 shows the main elements of the bug localization process used in this study. There are two main elements in this process: corpus creation and query formulation.

2.2 Corpus creation

Before the user can enter queries in the search engine, the software repository must be indexed according the chosen IR method. An important problem before the indexation process is the corpus creation. Figure 1 shows that corpus creation consists of corpus extraction and corpus filtering. In corpus extraction, the source code is parsed and the documents are extracted. Each document may be an entity of the source code (such as a class or a method). In this study, we have considered that a document will correspond to all terms of a method implementation, including the comments, because it seems reasonable for the developer to have an indication of which method(s) contains the bug. Other works that used the method implementation granularity [5, 8].

General processes do not filter the corpus, i.e., all the extracted corpus is searched. In this study, we will analyze if different alternatives of corpus filtering may have influence on the result of bug localization. The use or not of these filters can be considered a calibration parameter of bug localization approach. In this paper, two filters are studied, namely QTL and ET, and they will be explained in the section that describe the parameters investigated in this work.

2.3 The query formulation process

Other important element in the bug localization process is the query formulation. Bad queries in traditional web searches result in a bad list of results. Experienced users have

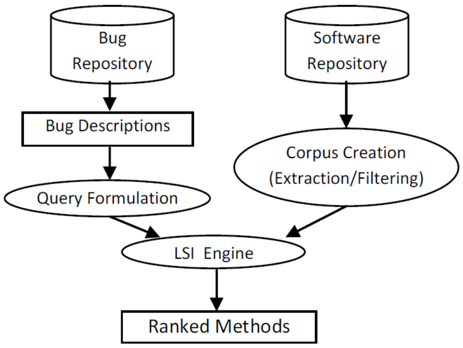


Figure 1: The bug localization process

success to get the desired results when using a web search engine because of their ability to choose better keywords. This is not different in the bug localization process. In order to reduce the influence of the query formulation process in this study, we devote the next subsection to define a systematic process to formulate the query. We neither advocate in this work that this query formulation process is perfect, nor that we will study the influence of different strategies of query formulation in the performance of the bug localization process. The goal of having a systematic process is just to have a uniform choice of query terms when searching for the localization of different bugs in different systems, so that the experiment can be reproducible. The input for a query formulation is the bug description available in the bug repository. Bug descriptions used in this study have a short description and a detailed description. Figure 3 shows an example of a bug description for ArgoUML in the first three lines of the table. Figure 2 shows the query formulation process. There filters are applied in the bug description in order to extract the term of the query:

1. **Filtering relevant sentences**. Leave sentences related to the failure reproduction. Sentences with the following content should be removed from bug description: name of the software, information of software and OS version, bug identification numbers, input data provided as example, error confirmation, redundant scenarios confirming that the error is actually present, preferences on behavior and other related errors. In the detailed description of Figure 3, two sentences were filtered in this step, because they have descriptions of a related error, a version number and a preferred behavior.
2. **Filtering classified terms**. Select and classify terms from the sentences previously selected into the following categories:
   (a) Where the failure occurs?
   (b) Which verb characterizes the action that triggers the failure?

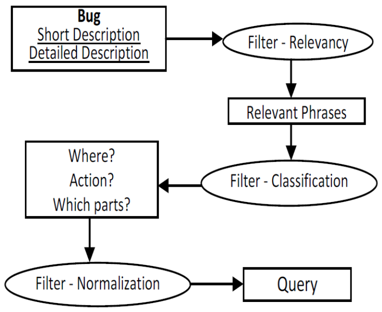


Figure 2: Query Formulation Process

(c) Which are the involved parts of the software?

In Figure 3, an example of this filter is shown in lines 4-6 of the table.

3. **Normalization of selected terms**. Express terms in singular form. Separate CamelCase terms preserving the original form. Separate compound words preserving the original one. Separate hyphenated terms. Transform to lower case. Correct typos. Maintain abbreviations. Remove repeated terms. In Figure 3, the final query resulted from this normalization is shown in line 7.

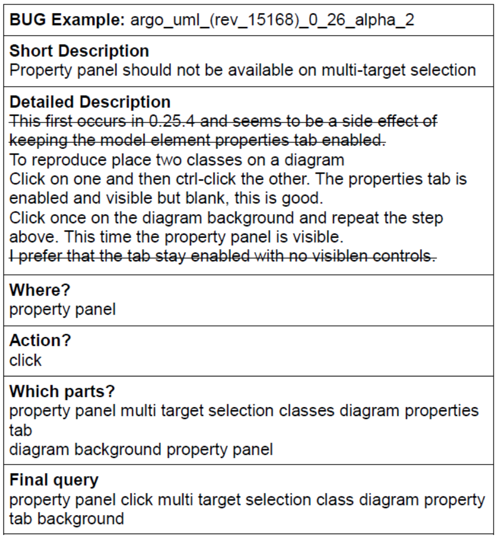
2.4 The Investigated Parameters

Some factors are known to influence the performance of the LSI method. In this study, we decided to select specific factors for maintenance tasks, such as different criteria for corpus creation, in combination with general factors, such as, the use of stop-words, weighting transformations for the term-document matrix and the dimensionality reduction parameter (k) for LSI.

Different criteria for corpus creation are defined as different filters to be applied in the corpus extracted from the entire source code. The result of filter application is the removal of some documents. Two filters for removing documents (methods) from the analysis will be studied:

1. **Execution Trace Filter (ET)** is a filter based on removing methods (documents) that do not occur in an execution trace that reproduce the bug.

2. **Query-term-like Filter (QTL)** is a filter based on execution trace and query terms. This filter enhances the previous filter removing all methods that have no relation with



BUG Example: argo_uml_(rev_15168)_0_26_alpha_2

Short Description
Property panel should not be available on multi-target selection

Detailed Description
~~This first occurs in 0.25.4 and seems to be a side effect of keeping the model element properties tab enabled.~~
To reproduce place two classes on a diagram
Click on one and then ctrl-click the other. The properties tab is enabled and visible but blank, this is good.
Click once on the diagram background and repeat the step above. This time the property panel is visible.
~~I prefer that the tab stay enabled with no visiblen controls.~~

Where?
property panel

Action?
click

Which parts?
property panel multi target selection classes diagram properties tab
diagram background property panel

Final query
property panel click multi target selection class diagram property tab background

Figure 3: An Example of Query Formulation

any of the terms of the query. Let t be a stemmed query term and t' be a method term. The criterion for a method having a relation with a query is that for some t of a query, it should be a case insensitive substring of some t' of the method.

The choice of the stop-words set will also consider the specificity of software systems and there will be three possibilities for the stop-words set:

1. **English language stop-words (ESTW)**. The words used in this study were collected from the Information Retrieval Group at University of Glasgow[2].
2. **Java language stop-words (JSTW)**. The words used in this study are the keywords of the Java language.
3. **(AllSTW)**. The union set of Java and English stop-words.

The study will consider some weighting mechanisms for the term-document matrix. Common weighting mechanisms are *tf* (term frequency) or *tf-idf* (term frequency-inverse document frequency). The proposed weighting mechanisms that will be studied are based on the intuitive idea that once the developer is trying to find the method where the bug is located, if

[2]http://ir.dcs.gla.ac.uk/resources/linguistic_utils/stop_words

he/she enters a query with the exact name of a method, then the corresponding method should be the best ranked one. So, terms that occur in names of methods and classes should have greater weight. These weighting mechanisms can be applied considering the term obtained from the class name (CN), from the several terms (fragments) that compose a class name (FCN), and the same for method names (MN and FMN, respectively). The weighting scheme adds to the term frequency, the value of the greatest frequency of a term in all documents multiplied by 2. This guarantees that this “weighted" term has higher relevance among the others in that document (method). The dimensionality reduction parameter (k) will be studied with common values adopted in the literature, such as, 100, 200 and 300 [7]. According to De Lucia and et al. (2007), the k value should be large enough to fit all independent concepts (a concept tries to cluster related terms to a document and tries to cluster related documents to a term) and small enough in order to do not fit unimportant details and sampling error. In one of their case studies, they studied k as 10 to 100% of the number of documents (150) and observed that for k > 30 (20%) the results are similar. In other case studies, they used k = 123 (=40% of 309 documents). ArgoUML has around 15000 methods and JEdit around 6000 methods. We preferred using 100, 200 and 300 because [Dumais, 1992] reported on good results for 20000 to 220000 documents using k between 235 and 250 concepts.

2.5 Research Questions

The following research questions are posed:

- RQ1: Concerning two different systems, ArgoUML and JEdit, is there any combination of parameters that would produce the best query results for both systems?
- RQ2: Are the proposed filters aiming at reducing the corpus worthwhile for both systems?

2.6 Experimental Setup for RQ1

Concerning RQ1, 10 bugs of the ArgoUML system and 10 bugs of the JEdit were selected to assess the influence of the parameter values. The bugs were selected randomly, but they should satisfy some restrictions, such that:

- they should be reproducible because the filters of the corpus require the reproduction of the bug to extract the trace information;
- the information about which methods were changed to perform the correction should also be available to evaluate the quality of the ranked list based on the position of these methods. This restriction has limited the number of bugs used in this experiment.

The corpus was indexed with the several combinations of the seven parameters. These parameters are the independent variables of the study. The possible values for each factor are $k = \{300, 200, 100\}$, CN = $\{0, 1\}$, FCN = $\{0, 1\}$, MN = $\{0, 1\}$, FMN = $\{0, 1\}$, STW = $\{ESTW, JSTW, AllSTW\}$, Corpus Filtering = $\{ET, QTL\}$. Indeed, there are 288 combinations of these values. For each bug, the query formulation process was applied and the query terms were collected. For each one of those combinations, the LSI engine was configured and the queries were processed.

The dependent variable is the accuracy of the result. In the case of using IR techniques for bug localization, the result will be a ranked list of documents which have better similarity with the query. As a matter of fact, the bug may be located in several parts of the program, so the ranked list of methods should have ideally the several methods with bugs located in the first positions of the list. Moreover, one approach to assess the accuracy of the ranked list of methods is to evaluate the position of all methods with bug in the list. However, another approaches [5, 8] argues that the IR technique should indicate just one method with bug and the user would find the other methods using the first method that effectively contained the bug. We consider that both approaches are valid and they will be studied in the evaluation.

The bugs for ArgoUML used in this study were extracted from argouml.tigris.org bug/issue repository. The extracted issues for ArgoUML and the corresponding new revisions were: issue 5038: revision 15156, issue 5280: revision 15501, issue 5266: revision 15427, issue 5115: revision 15145, issue 5160: revision 15090, issue 5202: revision 15169, issue 5229: revision 15234, issue 5387: revision 15689, issue 4430: revision 15750 and issue: 5591: revision 16518. The bugs for JEdit were extracted from the *Tracker* repository at sourceforge.net. The IDs of the bugs were 2173087, 2819438, 2823909, 1538720, 1534016, 2089713, 2946203, 2045733, 2781716 and 2881152 .

The strategy to answer the RQ1 was conducted as follows. For each bug of each system and for each possible combination of parameter values, a ranked list of methods was produced after querying the corpus with corresponding input query defined for each bug.

In each ranked list, the methods containing the bugs were found using the bug fixing information. Ideally, the methods with bugs should be in the first positions of the list. Considering that the developer would try to find the bug analyzing the first method in the list, and then the second method, and so on until the bug is found, the strategy to find the best combination was computing, for each combination, the sum of methods necessary to be checked in order to find all 10 bugs using a specific combination of parameters.

Considering that the answer to RQ1 is that there is no best combination that could be used equally for both systems, we could further investigate how the parameters behave together. In this case, we perform other analysis to identify a combination (or a set of combination) that are reasonable to use in most of the cases, or that the software engineer could

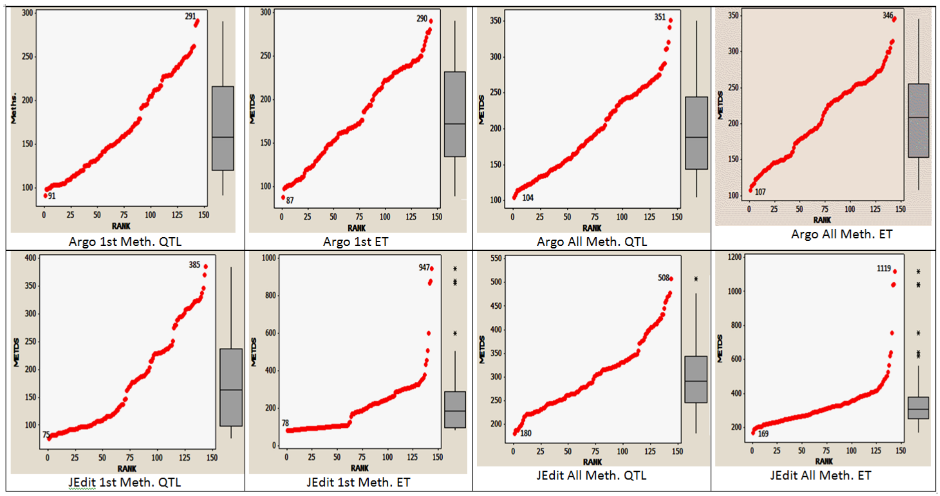


Figure 4: Scattering of Number of Methods (Precision) per Rank Position

use as a default combination for those systems. This analysis will use two strategies based on a predominance analysis and on the overall accuracy of the combination. The definition of the default combination for both systems will use the results of both strategies. The first strategy isolates each parameter and verifies which values for it produce a better result for one combination of the other parameters. The parameter value that produces more better results than the other values is the predominant value for that parameter. In this strategy, we compare how each possible value of a parameter behaves with all possible combinations for the other parameters. For each combination of the other parameters, we have varied the value of the studied parameter and calculated which value is winner for each combination. This winner parameter value is the one which requires fewer methods to be analyzed considering the corresponding ranked list. All methods with bugs were considered in the calculation. The strategy counts how many times a specific parameter value wins and the **predominant parameter value** is the one that wins most.

The second strategy is based on the overall analysis of the total number of methods required to check for a bug in the ranked list. The difference between these strategies is that the first does not take into account on how much a parameter value wins, but only in how many times it wins. Since the second strategy count all methods required to check for a specific parameter value, it will balance on how much a parameter value is good or not.

2.7 Experiment Setup for RQ2

The analysis of the corpus filters ET and QTL was compared with the use of no filtering mechanism. This is the independent variable.

To evaluate if the proposed filters for reducing the corpus are worthwhile, we proposed the analysis of the following dependent variables:

- execution time for indexing and searching;
- accuracy of the ranked methods.

To evaluate these variables, 3 bugs of JEdit and 3 bugs of the J text editor for Armed Bear Common Lisp[3] , J for short, were selected. The bugs for J also were extracted from the *SourceForge Tracker*. A limiting factor for the choice of these bugs was the necessity to index the whole source code to compare with the filtered source code. ArgoUML was not used in this study because it was not possible to index the whole system using the LSA package of the statistical software R. The trial execution stopped when the memory allocation reached the maximum of 7932Mb available on a desktop computer. So, this study was carried out with the JEdit and ArmedBear-J. The JEdit bug IDs used in this study were 2173087, 2819438, 2823909 and for J were 1175869, 648046, 761918.

3 Results

3.1 RQ1: Analysis of the Best Combination

The combinations were ranked and the best ones are shown in Table 1. In other words, for ArgoUML, considering we were searching for only one method with bug (column *Target*) and the filter QTL was applied, the configuration that resulted in the lowest number of methods necessary to check in order to find a bug is $k = 200$, CN = 1, FCN = 0, MN = 0, FMN = 0 and STW = All. So, this was the best configuration and the mean number of methods per bug necessary to analyze was 9.1. In fact, Table 1 is a simplified view of Figure 4 concerning the best combination of parameters. The quality of combination of parameters is measured with the sum of the number of methods necessary to check to localize all studied bugs of a system. For instance, in order to find a method that contain a bug for all 10 bugs studied for Argo using the QTL filter (Figure 4, left-upper hand side), the minimum number of methods necessary to check were 91 methods, using k=200, CN=true, FCN=false, MN=false, FMN=false and STW=All. In other words, for ArgoUML, considering we were searching for only one method with bug (column *Target*) and the filter QTL was applied, the

[3]http://armedbear-j.sourceforge.net

System	Target	Filter	K	CN	FCN	MN	FMN	STW	#Meths.
Argo	1^{st} Meth.	QTL	200	●	○	○	○	All	91
		ET	300	●	○	○	○	All	87
	All Meth.	QTL	300	●	○	●	○	All	104
		ET	300	●	○	○	○	All	107
JEdit	1^{st} Meth.	QTL	300	●	●	●	●	Java	75
		ET	300	●	●	●	●	All	78
	All Meth.	QTL	300	○	●	●	●	Java	180
		ET	300	○	●	●	●	All	169

Table 1: Parameter Values of Best Combinations

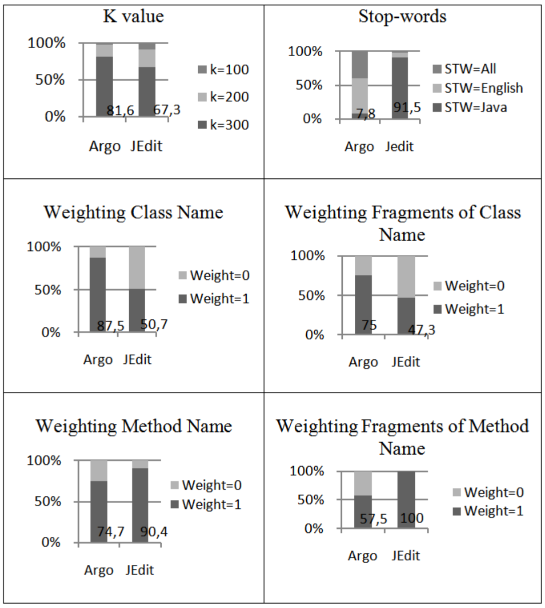
configuration that resulted in the lowest number of methods necessary to check in order to find a bug is k = 200, CN = 1, FCN = 0, MN = 0, FMN = 0 and STW = All. So, this was the best configuration and the average number of methods per bug necessary to analyze was 9.1. With the other combinations of parameters, it was necessary to check more methods, and with the worst combination of parameters it was necessary to check 291 methods (could be seen in Figure 4).

Figure 4 shows the scatter-plot (with a box-plot) of the number of methods necessary to analyze to find the bug considering all analyzed bugs of ArgoUML and JEdit. Notice that the best combination for ArgoUML using QTL filter required 91 methods to find the 1st method with bug, considering the 10 bugs, and the worst combination required 291 methods. This result shows the importance of choosing adequate parameters to calibrate the technique because choosing inadequate parameters may degrade considerably the accuracy of the query result. We can observe a linear degradation in almost all situations.

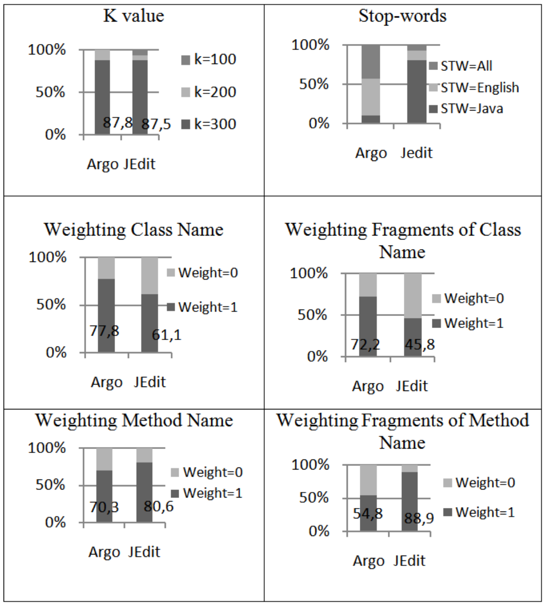
The **first part of first research question** can be straightforwardly answered negatively with the results shown in Table 1, because there is no consensus for the parameters values. The value for k was predominantly 300, but k=200 produced better results in some conditions. Using the weighting function CN predominantly produced the best results, but for JEdit and considering we want the localization of all methods with bugs, the best results were achieved when CN was not used. For ArgoUML, the absence of the weighting functions FCN, MN and FMN predominantly produced the best results, but for JEdit these weighting functions produced the best results.

3.2 Predominance analysis - parameter values that wins most

In this analysis, the best parameter values are the ones that have won more than others. Figure 5a shows the results of predominant values considering QTL corpus filtering and Figure 5b shows the results considering ET corpus filtering. The results are reported below.



(a) QTL Filtering



(b) ET Filtering

Figure 5: Predominance of Parameter Values

For the *dimensionality reduction parameter*, we observed that k=300 have higher predominance regardless the studied system and regardless the corpus filtering approach. Indeed, k=300 had better performance with filtering using only execution traces (ET) because this approach has a less aggressive filtering mechanism and it is reasonable to argue that greater k works better with a greater corpus.

For the *stop-word parameter*, we observed that the studied systems impose important influence in the selection of the stop-word criterion. JEdit's results with Java stop-words were much better and highly predominant. In ArgoUML, Java stop-words had a very poor result, and the use of English stop-words is preferred with this system.

For the *class name weighting parameter*, we observed that the use of weight was predominant only for ArgoUML. For JEdit using this weighting criterion was indifferent, because the absence of weighting also was not predominant. So, we could recommend the use of weight function parameter.

For the *class name fragments weighting parameter*, the observation was almost the same as the class name factor. Indeed, even for ArgoUML the weighting function parameter is not as strong as using the class name weighting function, and for JEdit, it has won in 47,3% using QTL filter and 45.8% using ET filter. We could recommend the use of this weight function for ArgoUML but not for JEdit.

For the *method name weighting parameter*, the observation was that its use has predominantly better results in both systems. Interestingly, JEdit, which did not have much influence with class name weighting, is highly influenced by the method name weighting function, even more than ArgoUML. We could recommend the use of this weight function for both systems.

Finally, for the *method name fragment weighting parameter*, we also observed that weighted method names fragments have high predominance in JEdit, even more than method name weighting. However, for ArgoUML, weighted method name fragments were only slightly predominant. We could recommend the use of this weight function, strongly for JEdit and weakly for ArgoUML.

Concluding, if we consider predominant parameter values that have more than 50% of better results, independently of the system and filter, then we could indicate a predominant better configuration for this study: k=300, CN=1, MN=1,FMN=1.

Indeed, we can see in Table 1 that this combination gives the best result for JEdit when considering the localization of only the first method. However, as we have already commented in the beginning of this section, this is not the best combination in all cases.

Table 2: General Ranking for JEdit and Argo + QTL and ET + First and All Methods

(a) Ranking

RANK	**K**	**MN**	**FMN**	**STW**	**CN**	**FCN**	**Meths.**
1	300	1	1	3	0	1	1026
2	300	1	1	1	1	1	1040
3	300	0	1	3	1	1	1042
4	300	1	1	2	1	1	1067
5	300	0	1	1	1	1	1069
6	300	0	1	3	0	1	1069
7	300	0	1	2	1	1	1081
8	300	1	1	1	0	1	1101
9	200	0	1	3	0	1	1103
10	200	1	1	3	0	1	1110
11	200	0	1	3	1	1	1131
12	300	0	1	1	0	1	1133
13	300	1	1	2	0	1	1135
14	300	0	1	2	0	1	1152

(b) Correlation of parameters

	K	MN	FMN	CN	FCN
PM	0.00				
	1.00				
PFM	0.00	0.00			
	1.00	1.00			
PC	0.00	0.00	0.00		
	1.00	1.00	1.00		
PFC	0.00	0.00	0.00	0.00	
	1.00	1.00	1.00	1.00	
Meths.	-0.37	0.01	-0.39	-0.16	-0.31
	0.00	0.91	0.00	0.056	0.00
Cell Contents:	Pearson correlation				
	P-Value				

3.3 The parameter values that produce an overall lower effort

In this strategy, we have produced a general rank for the combination of the parameters: dimensionality reduction, weighting functions and stop-words. There are 144 possible combinations for these values. There are three variables to analyze each combination: system (ArgoUML or JEdit), target of search (only the first method with bug or all methods with bug), corpus filter (QTL or ET). In Table 2a, we show the rank of the 10% best combinations. The rank is computed based on the column *Meths.*, which the sum of all number of methods returned by all possible variations of each combination (system, target and filter). So, this table represents an overall result for each combination. It is interesting to note that the

Table 3: Ranking for **JEdit or Argo** + QTL and ET + First and All Methods

(a) Correlation of parameters - ArgoUML

	K	MN	FMN	CN	FCN
Meths.	-0.46	-0.12	0.18	-0.24	-0.49
	0.00	0.15	0.03	0.004	0.00

(b) Correlation of parameters - JEdit

	K	MN	FMN	CN	FCN
Meths.	-0.18	0.08	-0.57	-0.06	-0.09
	0.03	0.34	0.00	0.51	0.31

Table 4: Ranking for JEdit/Argo, QTL/ET, **First Method or All Methods**

(a) Correlation of parameters - First Method

	K	MN	FMN	CN	FCN
Meths.	-0.31	0.06	-0.45	-0.19	-0.32
	0.00	0.49	0.00	0.02	0.00

(b) Correlation of parameters - All Methods

	K	MN	FMN	CN	FCN
Meths.	-0.42	-0.04	-0.32	-0.13	-0.29
	0.00	0.66	0.00	0.12	0.001

best ranked combination is similar to the proposed combination in the previous section, i.e., k=300, CN=1, MN=1,FMN=1.

Nonetheless, we conducted a Pearson correlation analysis of the parameters values among themselves and especially with the column *Meths.*, which result is shown in Table 2b. The results have shown that the parameters are not correlated with themselves (P-value $= 1.0$). A negative correlation indicates that a higher dimensionality reduction or the use of a weighting function is correlated with lower number of methods to be checked. A statistically significant correlation should have P-value ≤ 0.05. The parameters that had significant correlation were k and FMN, which are in the predominant combination shown in the previous subsection. Interestingly, although the use of the weighting function MN has predominantly won, there was no correlation of their use with fewer methods to be checked. CN have correlation at P-value $= 0.056$, but rigorously the common accepted value should be P-value ≤ 0.05. For our proposes, we argue that it is reasonable to accept this value because this is just one approximation of which *default* value should be used, and not a value that should strictly obey a statistic rule. Other interesting result is that the weighting function FCN, although not predominant, has a significant negative correlation. A possible cause is that the predominance analysis did not take into account the difference on the number of methods when winning or losing.

Table 3a and 3b are useful to identify if there are significant differences between the variables *System*. Table 3a shows that there is a significant correlation of parameters k, FMN, CN and FCN. This result is consistent with the general case, except that was found a negative correlation for the weighting function CN indicating that the use of CN produces a better result for ArgoUML. The comparison with the predominance analysis shows equivalent results for k, FMN and CN for the general case, and specifically for FCN for the case of ArgoUML. Interestingly, the predominance of MN did not reflected into a negative correlation as we could expect. Table 3b shows that there is a significant correlation of only of parameters k and FMN. This result is consistent with the general case, except that was found no correlation for the weighting function FCN indicating that it is not recommended the use of FCN for JEdit. The weighting function MN did not correlate negatively as we could expect from the predominance analysis. The weighting function CN did not correlate, but it was borderline in the predominance analysis for JEdit.

Table 4a shows that there is a significant correlation of parameters k, FMN, CN and FCN. This result is consistent with the general case, except that was found a negative correlation for the weighting function CN indicating that the use of CN produces a better result for the developer is looking just for the first method. Table 4b shows that there is a significant negative correlation of parameters k, FMN and FCN. This result is fully consistent with the general case.

3.4 RQ2: Analysis of Corpus Filtering

Despite of the fact, that we could not index the whole corpus of ArgoUML, we have accounted the number of documents and terms that were filtered using ET and QTL filters. Figure 6 shows the number of documents of ArgoUML and JEdit, both before and after applying the respective filters. The results show a significant reduction in the number of documents after filtering. Figure 7 shows a similar behavior for the number of terms. These results suggest that the obtained reduction would have an important impact in the final result of the queries. Figure 8a shows the number of documents of the corpus of JEdit and J for the revisions corresponding to each bug analyzed. The number of documents for different bugs of the same system is different because the execution traces used in the filtering mechanisms are different for each bug. So, since each document corresponds to a method, the called methods are different from one bug reproduction to another. The results are shown when no filter was applied, and when the ET and QTL filter were applied. It is important to note that the plot is in logarithmic scale showing a major reducing of the size of corpus using the execution trace filters.

Figure 6: Number of Documents: before filtering, after ET filter and after QTL filter

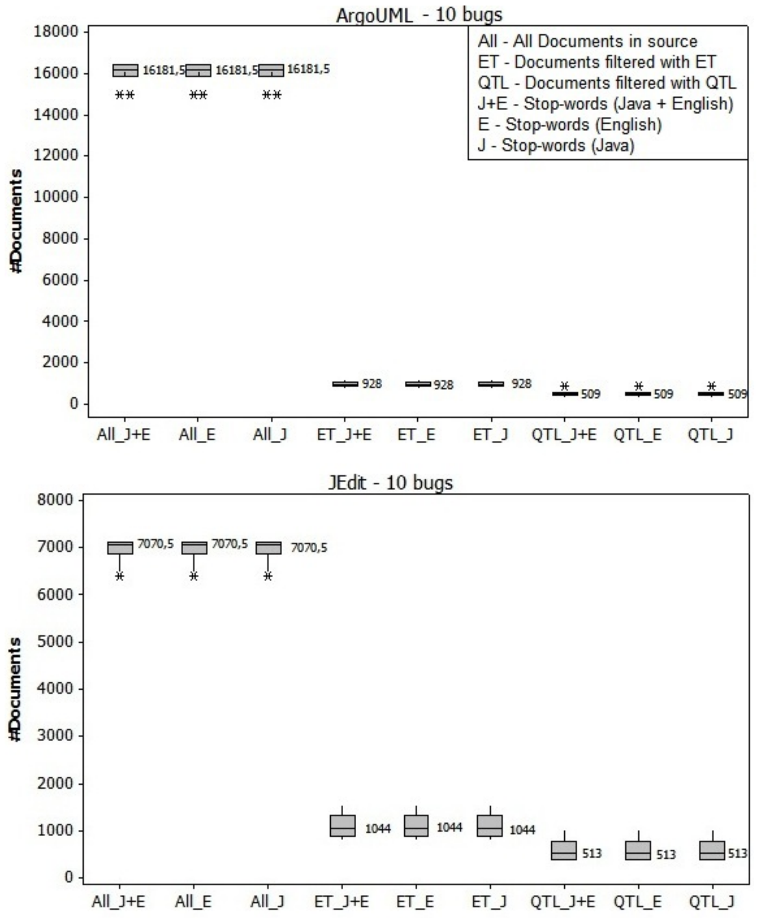
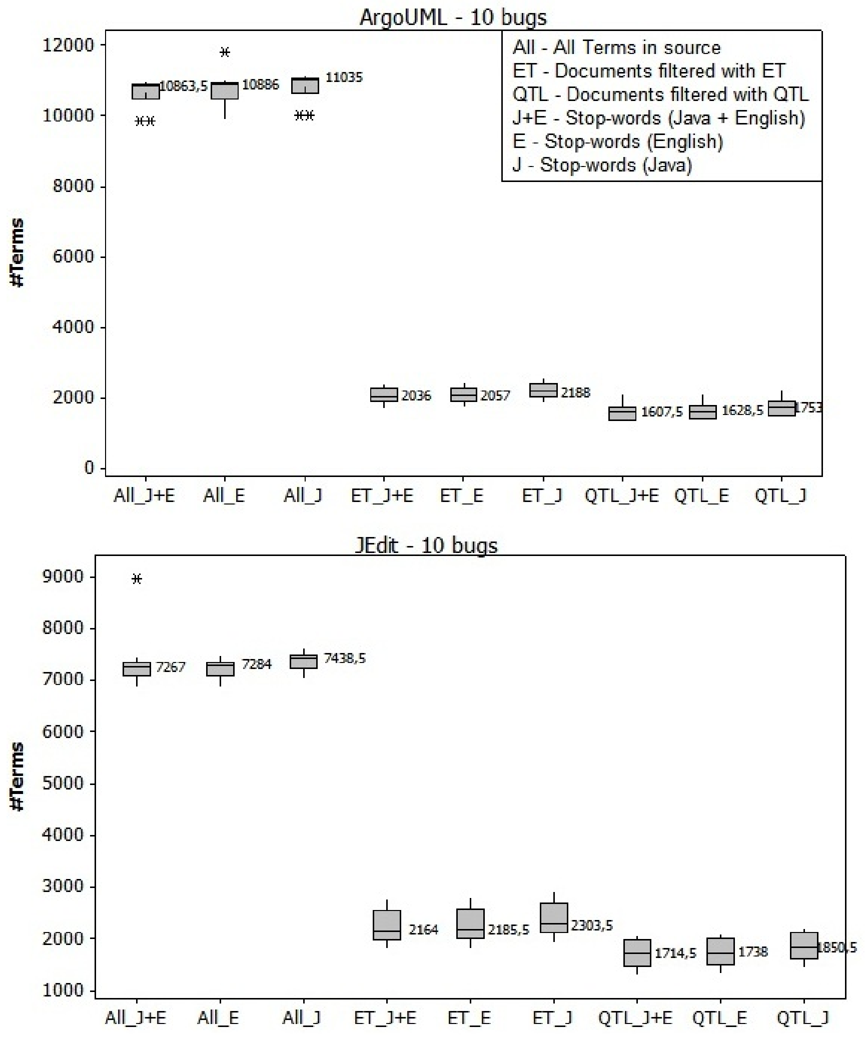
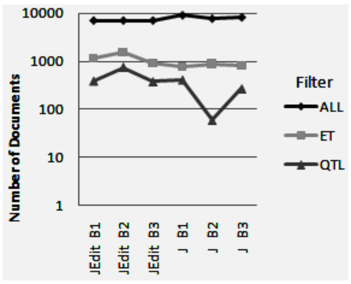
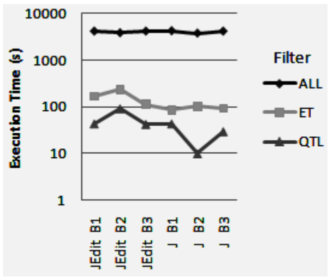


Figure 7: Number of Terms: before filtering, after ET filter and after QTL filter

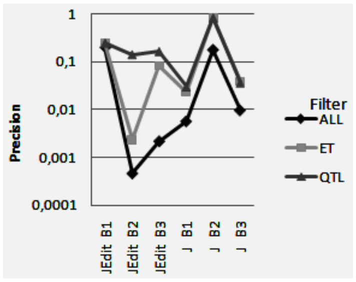




(a) Number of documents in the corpus



(b) Elapsed time to index and process



(c) Precision of the queries - J and JEdit

Bug	ALL	ET	QTL
JEdit B1	0.2	0.25	0.25
JEdit B2	4.65E-04	0.002	0.143
JEdit B3	0.002	0.083	0.167
J B1	0.006	0.023	0.030
J B2	0.177	0.833	0.833
J B3	0.010	0.039	0.037

(d) Precision values

Figure 8: Results of Corpus Filters

Figure 8b shows the execution time to index the respective corpus using the same filters. It is also shown in logarithmic scale. There is a strong correlation between the size of the corpus and the execution time.

Figures 8c and 8d shows the result of precision of the query results for different filtering approaches. The values were calculated using average precision [12]. It was expected that the reduction of the corpus could enhance significantly the precision. Indeed, the results have shown that the filters provide better precision in all cases. There is an important gain in all cases, except for the bug 1 of JEdit, where the precision of ET and QTL was 0.25 and the precision of *no filtering* was 0.2. Although, the results have shown that the QTL Filter had produced equal or better precision, the difference was mostly significant for bugs 2 and 3 of JEdit.

4 Discussion

Concerning RQ1, the results presented previously have shown that different systems may require different parameter values to get better accuracy. This situation suggests that the software engineers should reproduce similar analysis as ours to find which parameter value is best suited to their system. However, this may be a laborious task that may hinder the adoption of IR engines. The results also have shown that different analysis strategies can produce different conclusions. So, we decided to provide the intersection of both strategies to indicate a confident default recommendation. In the Table 5, we provide a pictorial recommendation of each parameter value that software engineer may use by default and still get good results. For K and FMN, the result is unanimous. The use of CN and FCN is strongly recommended for Argo but also is borderline for JEdit, so we decided to recommend it by default, although their signifance is inconclusive. The use of the weighting function MN was also inconclusive.

Concerning RQ2, the results of this study show how the corpus filtering has influence in the execution time to index and process the corpus and in the precision of the query results. This seems to be an obvious result, however it is important to note that in the case of filtering approach it should be expected some degradation in recall. The interesting result is that the proposed filters did not significantly affect the recall, except in one case: the bug 3 of J has 6 methods that needed to be fixed. After applying the QTL filter, one of these methods has been discarded. For all other 22 bugs analyzed, the ET and QTL filters removed no method on which the bug was located, i.e., yielding a 100% recall.

Moreover, the QTL filter consistently had better performance than the ET filter.

Our conclusion suggests that based on the previous results the use of QTL or ET filters is strongly recommended against the use of no filter.

Another point of discussion is that other approaches [5, 8, 2] have applied the filters

Table 5: Default Recommendation

Strategy	System	**K**	**CN**	**FCN**	**MN**	**FMN**
Pred. Analysis	Argo	200	●	●	●	●
	JEdit	300	◐	○	●	●
Overall Effort	Argo	300	●	●	○	●
	JEdit	300	○	○	○	●
	Both	300	●	●	○	●
Default Recommendation		300	●	●	-	●

after ranking the documents with the IR engine. However in our approach, we decided to perform a different strategy: apply the filters before indexing the documents with the IR engine. Apparently, the hypothesis that this kind of ordering seems to be invalid because it has been recognized that the larger the collection of documents, the better accuracy achieved in the queries. In fact, our results have shown that this situation is **not necessarily** true and our *nip-in-the-bud* strategy should be considered in a customized analysis of the system.

5 Threats to Validity

One threat is related with the size of our sample. We considered 20 bugs from 2 different systems. As a matter of fact, a large number of bugs would reduce the bias caused by the variation caused in the query formulation process. To alleviate this bias, at least we have proposed a systematic way to extract the query terms to guarantee some fairness among the queries of different bugs, but still the choice of terms is a threat to validity. In fact, although a larger sample would provide more confidence in the results, the used sample was sufficient to show that there is no best combination of parameters values that can be used in any case, even for systems in a similar domain. The analysis of the proposed corpus filters also suffer from this limitation because ArgoUML bugs could not be used. The experiment was run using the LSA package of the R statistical software. We had no control on the implementation of this package, and we cannot guarantee that a native implementation of LSI would provide better results on performance and index the whole ArgoUML corpus. However, the results on the performance and precision of the corpus filters suggest a plausible confidence despite of the small sample. The respective results on recall are a bit more sensitive in this case.

Other threats are: 1) the query formulation process is not entirely automated and thus may introduce the experimenter bias; 2) the studied systems were only written in Java; 3) the vocabulary of the source code may dramatically influence the experiment.

6 Threats to validity.

Even with the careful planning and execution of the experiments, some factors should be considered in the evaluation of the results validity.

6.1 Internal Validity

- Individual differences among the participants. The actions to minimize these differences were the analysis and classification of the capability of each participant to generate fairly balanced groups and also a random selection of which group the participant should be included.
- The participants' understanding about the dynamic approach may vary during the training. We tried to minimize this using the same printed material for everybody and practicing with the tools and approach before the experiment session.
- Subjectivity degree in the definition of which code elements are relevant for an activity. The result of some maintenance activities does not have a unique and correct answer. Developers may propose different, but correct results for the same activity.
- Inexperienced participants concerning the proposed approach. The comparison of the performance of the *Control* group, which had a familiar use of the traditional Eclipse-based approach with the performance of groups that do not have the long term experience of using a new approach may not be totally fair. We can consider that the results could be even better for *Traces* groups.

6.2 External Validity

Some other factors limit the generalization of the results like:

- The number of participants in each group for an activity. As in any statistical study, the larger and more representative is the sample, the stronger are the evidences. In fact, if the results were only analyzed separately for each activity, then this would be more sensitive. However, the results were analyzed considering the set of 4 tasks. This setting accounts for 12 observations for each group and 36 observations for the whole study. Moreover, the impact of the use of trace information (*Simple* or *Enhanced*) were analyzed with 24 observations. These numbers are significant considering related work. In [13], the authors show that only 6 out of the 114 selected papers in the above area were published using empirical evalution based on human subjects. From these 6 papers, only Quante's work has used more than 9 human subjects in the experiment [14]. In Quante's work, 25 students have participated in the main experiment. In our work, 27 developers have participated in the main experiment, so in this aspect, our work seems to be a clear contribution to this field compared to the state-of-art work.

- The representativeness of target systems. Although the system used in the first task is a home made system, JHotdraw and ArgoUML are well-known systems widely used in other empirical studies. Indeed, the number and domain of the used system limit the generalization of our results.
- The selected maintenance tasks. Although the maintenance tasks were chosen to be quite different from each other, it is not possible to generalize for the whole universe of possible maintenance tasks. Nonetheless, our results have shown a reasonable level of variability in the tasks results that enabled the qualitative discussion of important points.
- Only the Java programming language and Eclipse development environment were considered in this. Some of our results would be different if other languages and environments were used. For example, different languages may support different types of dependencies; different development environments may summarize and present code in different ways.

6.3 Construct Validity

Concerning our measurement framework there some issues to be pointed:

- The experimenter also has defined the dynamic approach being evaluated.
- An internally developed execution trace approach was used to represent the concept of using execution trace information. Since, there are no widely recognized tool support for this kind of approach, any other adopted solution would incur in similar threat. We tried to minimized this issue using well known tools and concepts during the definition our the studied approaches.

7 Related Work

Information retrieval methods have been largely applied in software maintenance activites that benefits from the recovery of traceability links between different software artifacts [15, 16]. This activities include bug localization, feature/concept location and developer identification.

Bug localization is a special problem of finding traceability information [17] from external artifacts of the software and internal artifacts, such as the source code [1, 10, 11, 5, 18, 19], because we want to recover a trace from the bug description artifact to the source code that encompasses the bug. Another software maintenance activity that is similar to bug localization is feature (concept) location [20] and several solutions based on information retrieval are applied to this problem [21, 22].

Antoniol et al. [10] have shown that probabilistic and vector IR methods provides a feasible technique to recover traceability links semi-automatically between code and documentation. They demonstrate the benefits of the IR engines against *grep brute force*. Marcus and Maletic [3] have shown that LSI performs at least as well as Antoniol's previous methods, and still has some benefits in source code processing. Marcus et al. [6] also apply LSI to concept location. This paper focuses especially on the analysis of semi-automatically generated queries, which is possible because of the ability of LSI indicate similar terms of the source code for a user defined term. This paper also have demonstrate benefits of LSI against *grep*.

Recently, De Lucia et al. [4] found evidences that the IR engine is an influence factor on the retrieval accuracy achieved by the software engineers. Hayes et al. [18] propose enhancements to a classical vector space model engine. The enhancements are basically based on introducing *key-phrases* (compound terms) into the vocabulary and introducing a thesaurus that allows some level of similarity between terms. As a matter of fact, the LSI approach naturally provide a kind of “latent" similarity and in our approach when we use the both method and class names and also their fragments, we are taking into account some form of compound terms.

Lukins et al. [8] studies LDA (latent Dirichlet allocation) as a new alternative for the IR engine. They argue that LDA provides better performance results than LSI because the number of methods to encounter the first method with bug is systematically smaller using LDA than using LSI. In fact, the results provided in this paper could also be used as hypothesis and analyzed for LDA. In our study, we also provided a systematic manner to extract the query terms from the bug description. The bug query may have a dramatic influence on the quality of the results, in the same way that bad queries have a similar influence when obtaining information from search engines in the web.

The use of dynamic information has been extensively applied to problems of program comprehension [13]. Execution traces constitute one of the most important source of information for dynamic analysis. In particular, the problem of feature location has largely benefit of the analysis of execution traces. A seminal work that relates the analysis of execution trace and feature location is *Software Reconnaissance*[23]. Some works have introduced filters based on execution traces to enhance IR-based approaches for feature location. Poshyvanyk et al. [5] have proposed PROMESIR combining Statistic Probabilistic Ranking - SPR and LSI. They combine the LSI and SPR rankings of source code elements. The SPR approach is based on execution traces captured during the execution scenarios associated with a feature.

The approach proposed in this paper does not produce other ranking than the LSI ranking, but use the execution scenarios to filter the corpus. In some sense, both have the same intent to use the execution scenarios to increase the accuracy of the approach. Moreover, our filtering approach also has the benefit of improving performance with corpus reduction. Their paper does not report on special weighting functions.

Several techniques have been proposed to enhance the results of LSI for bug localization and feature location. Sheng [24] proposed an enhancement to LSI based on the information of the static call graph extracted from the source code. They have shown improvements, but still they have introduced additional parameters to the technique. Another work that use different techniques on execution trace information to enhance the result of feature location is based on data fusion and web mining techniques [25]. They propose the additional filtering of IR based ranks using web mining algorithms, such as HITS [26] and PageRank [27], to further eliminate methods that are present in all features or are highly specific to a feature. Those algorithms are executed on graphs derive from the binary or frequency of method calls in the execution trace. Although the idea of using special filters to enhance the IR technique is not new [2, 5, 28], it is important to note that the focus of those work was not to study the characteristics and the impact of the use of different combinations of parameters, weighting functions, stop-words or corpus filters. We have shown that there is not necessarily one best parameter value, regardless the other parameters.

8 Conclusions

In this paper, we have presented a study on how the parameterization of the LSI technique for bug localization can influence its performance and accuracy. We designed an experiment to analyze several possible combinations of input parameters and observed that there is no best combination of parameter for general use. Moreover, we recommended a configuration that can be used as the default option for bug localization tools based on LSI approach in the cases where the software engineering cannot perform some experimentation to calibrate the parameter values for a specific system (see Table 5).

Specifically, we have shown that for JEdit a weighting mechanism for terms of method names would be an important decision, while a weighting mechanism for terms of class names does not have a major influence. For ArgoUML, the terms of class names have an important influence. As a matter of fact, we can suggest that in JEdit method names are more semantically aligned to the user vocabulary than class names. In ArgoUML that class names are more aligned than the method names. Although, the literature generally already reinforces the importance of meaningful identifiers in the source code, our results also reveal the special importance of careful choice of class and method names in the implementation when the proposed weighting functions are applied.

Another conclusion is that corpus filtering mechanisms also have an important influence on the performance and accuracy of the LSI approach. Indeed, we have shown that the proposed approach to filter the corpus based on both 1) the execution traces related to the bug reproduction (ET), and 2) the regular filter of documents using query terms (QTL) has shown important benefits for the LSI approach.

Moreover, as future work the proposed weighting functions and filters can also be applied to other IR engines, and special mechanisms that helps the software engineering to tailor the query could be incorporated as well.

References

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 40, Washington, DC, USA, 2000. IEEE Computer Society.

[2] Giuliano Antoniol and Yann-Gael Gueheneuc. Feature identification: An epidemiological metaphor. *IEEE Trans. Softw. Eng.*, 32:627–641, September 2006.

[3] Jonathan I. Maletic Andrian Marcus. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *In. 25th International Conference on Software Engineering (ICSE)*, number 3, pages 125–137, Portland, Oregon, USA, 2003. IEEE Computer Society.

[4] A. De Lucia, R. Oliveto, and G. Tortora. Assessing IR-based traceability recovery tools through controlled experiments. *Empirical Software Eng.*, 14:57–92, February 2009.

[5] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions Software Engineering*, 33:420–432, June 2007.

[6] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[7] April Kontostathis. Essential dimensions of latent semantic indexing (LSI). In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS 07, page 73, Washington, DC, USA, 2007. IEEE Computer Society.

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

[9] Scott Deerwester. Improving information retrieval with latent semantic indexing. In *Proc. of the 51st Annual Meeting of the American Society for Information Science*, pages 36–40, 1988.

[10] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Software Engeering*, 28:970–983, October 2002.

[11] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(13), September 2007.

[12] David A. Grossman and Ophir Frieder. *Information retrieval: algorithms and heuristics*. Springer, 2004.

[13] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684 –702, 2009.

[14] J. Quante. Do dynamic object process graphs support program understanding? – a controlled experiment. In *Proc. the Intl. Conf. on Program Comprehension*, pages 73–82, 2008.

[15] David Binkley and Dawn Lawrie. Information retrieval applications in software development. In *Encyclopedia of Software Engineering*, pages 231–242. Taylor and Francis, 2010.

[16] David Binkley and Dawn Lawrie. Information retrieval applications in software maintenance and evolution. In *Encyclopedia of Software Engineering*, pages 454–463. Taylor and Francis, 2010.

[17] Balasubramaniam Ramesh, Curtis Stubbs, Timothy Powers, and Michael Edwards. Requirements traceability: Theory and practice. *Ann. Softw. Eng.*, 3:397–415, January 1997.

[18] J.H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Proc. 11th IEEE Intl. Requirements Engineering Conf.*, pages 138 – 147, 2003.

[19] M. Lormans and A. van Deursen. Can LSI help reconstructing requirements traceability in design and test? In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp. –56, march 2006.

[20] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[21] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Václav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *in Automated Software Engineering (ASE'2007)*, page 234243, 2007.

[22] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, April 2006.

[23] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping program features to code. *J. Softw. Maint.: Res. Pract.*, 7(1):49–62, 1995.

[24] P. Shao, N.A. Kraft, and R.K. Smith. Combining latent semantic indexing and call graphs to improve feature location. In *Proc. of the13th Annual IASTED International Conference on Software Engineering and Applications (SEA)*, pages 49–54, Cambridge, MA, USA, 2009.

[25] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 14 –23, 30 2010-july 2 2010.

[26] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999.

[27] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[28] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Improving IR-based traceability recovery using smoothing filters. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 21 –30, june 2011.