Improving Feature Location by Transforming the Query from Natural Language into Requirements

Raúl Lapeña, Jaime Font, Francisca Pérez, Carlos Cetina
SVIT Research Group. Universidad San Jorge
Autovía A-23 Zaragoza-Huesca Km.299, 50830, Zaragoza, Spain
{rlapena, jfont, mfperez, ccetina}@usj.es

ABSTRACT

Software maintenance and evolution activities are responsible for the emergence of a great demand of feature location approaches that search relevant code in a large codebase. However, this search is usually performed manually and relies heavily on developers. In this paper, we propose a feature location approach that, instead of searching directly into code from a natural language query as other approaches do, transforms a natural language query to a query that is made up of the requirements that are located as relevant. Furthermore, our approach limits the scope of the code search space by selecting only the code of those products that hold relevant requirements. We evaluate the overall effectiveness of our approach in the industrial domain of train control software. Our results show that our approach improves in 18.1% the results of precision with regard to searching directly into code, which encourages further research in this direction.

Keywords

Feature Location; Software Maintenance and Evolution; Families of Software Products

1. INTRODUCTION

Companies accumulate a vast amount of software that implements the features of its product family over the years. Each feature represents a functionality that is defined by requirements. Adding or removing features to software products, improving existing functionality, creating new products from existing features, and removing bugs, are common activities performed by developers during software maintenance and evolution. No maintenance activity can be completed without locating in the first place the code that is relevant to the specific functionality [9]. This identification of the initial location in the code is known as Feature (or concept) Location (FL) [4, 9].

Hence, there is a great demand for FL approaches that can help developers find relevant code in a large codebase. To find relevant code, textual analysis can be performed using Information Retrieval (IR) techniques [9, 1] such as Latent Semantic Indexing (LSI) [7], Latent Dirichlet Allocation (LDA) [5], and Vector Space Model [30]. These techniques are statistical methods used to find a feature's relevant code by analyzing and retrieving words that are similar to a query provided by a user. Several FL approaches have emerged [2, 18] that analyze the words used in source code and perform code search based on the text similarity between code snippets and a query. Nevertheless, these approaches often have unsatisfactory results [23] since the precision values accomplished reach only up to 25%.

To overcome this issue, other FL approaches [13, 34] that tackle the problem by refining the query using semantic similar words have been proposed recently. However, it was observed that automatically expanding a query with inappropriate synonyms may produce even worse precision values than not expanding the query [23].

In this paper, we propose a FL approach, with a level of indirection with regard to existing FL approaches, that searches for relevant code directly from a natural language (NL) query in a family of software products, which has a set of formalized requirements but lacks requirements to code traceability. Our approach transforms the NL query into a query that holds the requirements located as relevant according to the similarity of the words used. In addition, our approach limits the scope of the code search space by selecting only the code of the products that hold relevant requirements. In industrial environments, this limitation is especially important since product families may comprise thousands of products. Then, both the resulting requirements query and the code of relevant products are taken as input to search for relevant code.

We evaluate our approach to investigate its overall effectiveness in an industrial domain, more specifically in the railway domain. Our industrial partner, *Construcciones y Auxiliar de Ferrocarriles* (CAF), is a worldwide leader in train manufacturing that provided a software product family of a total of seven real-world trains, which comprise around 2940 requirements and 1193 KLOC. In addition, we compare our approach with a conventional FL approach that searches directly into code in order to evaluate its effectiveness in terms of recall and precision [30]. The results show that our approach outperforms the conventional FL approach since it improves an average of 18.1% the precision score. This result encourages us to further research in this area.

The remainder of this paper is organized as follows. Section 2 presents the overall structure of our approach.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '16, September 16-23, 2016, Beijing, China
© 2016 ACM. ISBN 978-1-4503-4050-2/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2934466.2962732

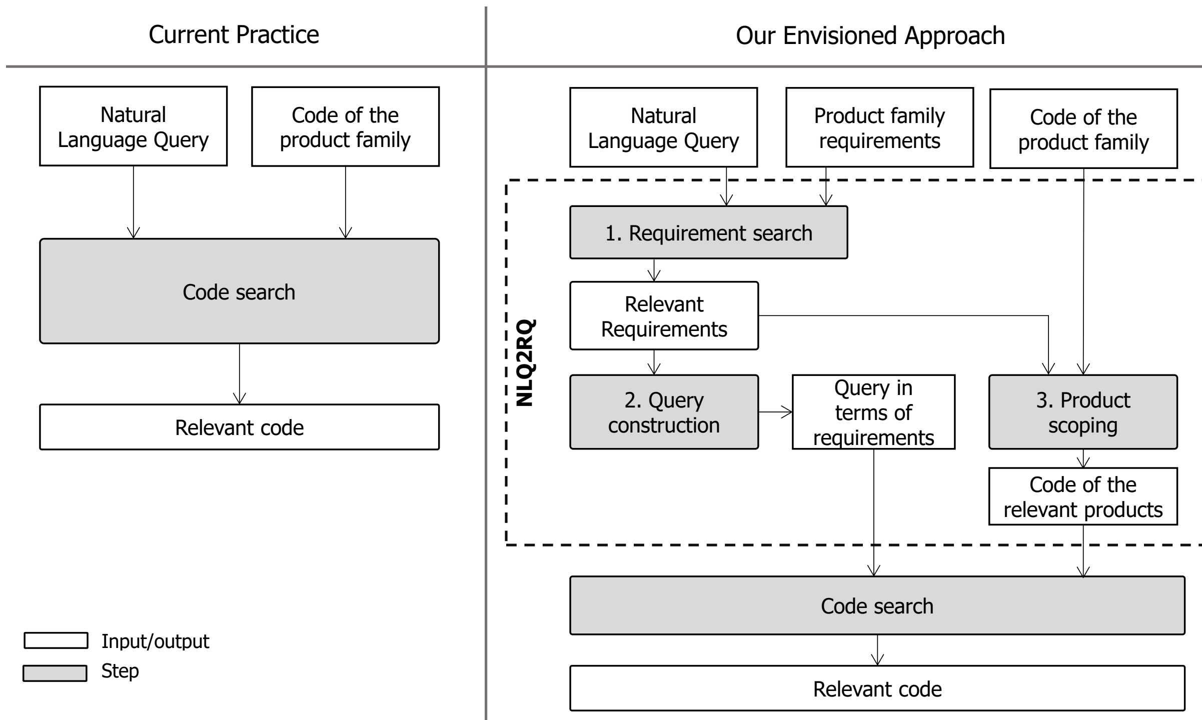

Figure 1: A highly simplified view of Feature Location: current state of practice (left) and our approach (right)

Section 3 describes the transformation from the NL query to the requirements query. Section 4 presents the experiments that we conducted to evaluate our approach. Section 5 discusses the results. Section 6 presents threats to validity. Section 7 reviews the related work, and Section 8 concludes the paper and outlines the future work.

2. THE OVERALL STRUCTURE OF OUR APPROACH

In this section, we present our approach to locate the relevant code for a natural language query. Figure 1 depicts a comparison between our approach and the current state of practice. In the current state of practice (see left side), both a query and the code of a product family are taken as input to directly search the code that is relevant. The code search can be done by analyzing the words used in the query and the source code. The idea is that the words encode domain knowledge, and a feature may be implemented using a similar set of words throughout a software system, making it possible to find a feature's relevant code textually.

In our approach (see right side of Figure 1), we propose to transform a natural language query to a requirements query (NLQ2RQ) by following the three steps that are shown in the figure (Requirement search, Query construction, and Product scoping). In the first place, we search for the requirements that are the most relevant to the natural language query. Afterwards, we construct the requirements query by joining the relevant requirements. Next, we limit the scope of the code search space by selecting only the code of those products that hold relevant requirements. Finally, we take as input both the requirements query and the code of the relevant products to search for the relevant code.

Our idea is that the words used in the NL query are closer to the requirements than to the code. Furthermore, we limit the code search space since only the code that holds the most relevant requirements is used, instead of using the code of the whole product family. This is especially important in industrial environments where a family of products can have thousands of products.

3. FROM NL QUERY TO REQUIREMENTS QUERY

In this section, we describe the three steps (Requirement search, Query construction, and Product scoping) that we propose in our approach. To illustrate the steps, we use a simple running example from our industrial partner, CAF[1], which develops software to control the high speed trains, regional and commuter trains, metros, trams and Light Rail Vehicles that they manufacture.

3.1 Requirement search

In order to search the requirements that are relevant to the natural language query, we perform a keyword extraction process and a textual analysis process.

3.1.1 *Keyword extraction*

Keywords of the requirements and the query are extracted and combined into a single set of terms in which duplicates are removed. To extract keywords, the analysis of POS tags is used for searching nouns and nominal structures, since they provide promising results in the extraction of keywords from technical documents [15, 22]. For example, the NL

[1]www.caf.es/en

query of our running example is 'doors in cleaning mode' and some of the extracted keywords are the nouns 'doors' and 'mode'.

Next, stemming [25] is used to reduce the extracted keywords to their root in order to unify words and avoid duplicity of terms. For example, 'doors' will be stemmed to 'door' and 'cleaning' to 'clean'. After, a filtering process is carried out to remove stopwords. Stopwords are non-important nouns and nominal structures provided by software engineers working in the train products (e.g., a stopword provided is 'Second'). For example, some keywords extracted from the query and requirements are: panto, clean, light, door, coupling, and brake.

3.1.2 *Textual analysis of requirements*

The keywords previously extracted are taken as input to perform an adapted LSI analysis [28] that obtains the most similar requirements of the product family by analyzing the relationships between *queries* and *documents* (bodies of text). We select LSI because most of the feature location research efforts show that its application provides better results [27, 26]. LSI constructs vector representations for both the *query* and *documents* by encoding them as a *term-by-document co-occurrence matrix.*

The upper half of Figure 2 shows an example of matrix for the LSI analysis in our running example. Each row in the matrix is an extracted keyword, whereas each column is a requirement (e.g. the R1 column represents the requirement 1 in product 1). Requirements are grouped by product for the sake of comprehensibility in the figure. The last column in the matrix is the NL query. Each cell in the matrix contains the frequency with which the *term* of its row appears in the *document* or *query* denoted by its column. For instance, in the matrix that is shown in the figure, the *term* 'door' appears three times in R1 from product 1, and once in Rx from product 1, Ry from product N, and the query.

After the matrix is fulfilled, the columns of the matrix are transformed into vectors. To do this, we normalize and decompose the matrix using a matrix factorization technique called *Singular Value Decomposition* (SVD) [19]. SVD is a form of factor analysis, or more properly the mathematical generalization of which factor analysis is a special case. In SVD, a rectangular matrix is decomposed into the product of three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed.

The lower half of Figure 2 presents the result of performing SVD over some requirements and the query in the three-dimensional graph provided. The vector named 'Query' represents the query column, and the other vectors represent some of the requirements columns. For example, the vector named 'R218.P3' represents Requirement 218 of Product 3.

To measure the similarity degree between the vectors, our approach calculates the cosine between the query vector and the requirement vectors. Cosine values closer to one denote a higher degree of similarity, whereas cosine values closer to minus one denote a lower degree of similarity. Similarity increases as the vectors point in the same general direction, i.e., as more keywords are shared between the requirements

Latent Semantic Indexing (LSI):

Requirements

Keywords	PRODUCT 1			...	PRODUCT N		Query
	R1	...	Rx	...	...	Ry	
PANTO	2	...	1	...	...	2	0
CLEAN	1	...	1	...	...	2	1
LIGHT	1	...	0	...	...	1	0
DOOR	3	...	1	...	...	1	1
CCU	0	...	1	...	...	0	0
COUPL	0	...	1	...	...	1	0
BRAKE	1	...	2	...	...	1	0
...	....	...	....	...	...	....	...

Singular Value Decomposition (SVD):

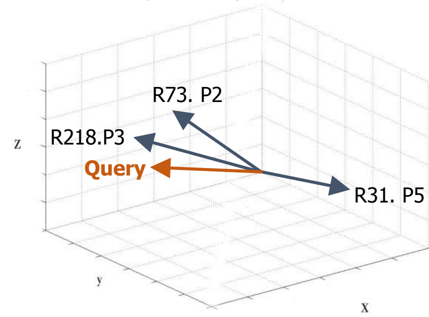

Figure 2: Analysis of requirements example

and the query. Having this measurement, our approach is then able to obtain a ranking with the most relevant requirements for the query. Following the running example, the vector that is the closest to the *query* vector is 'R218.P3', so it represents the most relevant requirement in product 3 for the query that is 'the CCU will enable the doors when the train enters in cleaning mode state'. The vector 'R73.P2' represents the second most relevant requirement to the query that is 'train speed will be limited in the washing state cleaning mode'.

At this point, it is important to note that among all the requirements identified as relevant, only those requirements that have a similarity measure greater than x must be taken into account. A good and widely used heuristic is x = 0.7. This value corresponds to a 45 degrees angle between the corresponding vectors. This threshold has yielded good results in other works [24, 29]. By following the running example, the vector 'R31.P5' is not taken into account as a relevant requirement because the angle with the query vector is more than 45 degrees wide. Determining a more generally usable heuristic for the selection of the appropriate threshold is an issue under study and further research is needed.

3.2 Query construction

To construct the query of requirements, we take as input

the most relevant requirements that we obtained in the previous step to join them since the performance is more effective with long and descriptive queries [31]. Specifically, we combine into a single query the relevant requirements by removing duplicates.

By following the running example, the most relevant requirement 'the CCU will enable the doors when the train enters in cleaning mode state', which was obtained in the previous step, is completely added to the requirements query. After, duplicates are checked between the requirements query and each word of the most relevant requirements. To do this, stemming is used to reduce the words to their root, unify words and avoid duplicity. From the second most relevant requirement obtained in the previous step 'train speed will be limited in the washing state cleaning mode', the words 'speed, be, limited, washing' are added to the requirements query since their root is not yet included. Hence, the requirements query obtained is 'the CCU will enable the doors when the train enters in cleaning mode state speed be limited washing'.

Several techniques such as [14, 20] can be used to construct the query selecting or removing the most frequent words. However, determining the performance of different techniques remains as future work.

3.3 Product scoping

In this step, we limit the scope of the code search space by selecting only the code of the products that hold the relevant requirements, which were obtained in the requirements search. In our example, the code from product 3 and product 2 is selected, since they hold the relevant requirements (R218.P3 and R73.P2, respectively). Hence, instead of searching code in the whole product family, we only search in the products that include relevant requirements.

Once the product scoping is done, we take as input the query in terms of requirements that was previously constructed and the code of the relevant products in order to perform a code search, obtaining a ranking of relevant methods (see bottom right of Figure 1).

4. EVALUATION

We perform an evaluation in the railway domain. In this section, we present our experimental setting, evaluation metrics, technological decisions, and experimental results.

4.1 Experimental setting

The product family that we use in our experiment consists of seven real-world trains, provided by our industrial partner. In total, the product family is defined by an approximate number of 2940 requirements and comprises near to 3850 methods, which account for around 1193.5 KLOC. Each train is defined by an average of about 420 requirements, which in turn have an average length of around 50 words. The trains are coded by an average of about 550 methods, each one having an approximate extension of 310 LOC. Therefore, each train in our software product family is coded in about 170.5 KLOC.

Natural language queries can come from textual documentation of the products, bug reports and oral descriptions from the engineers who work in our industrial partner. In this experiment, we use 21 natural language queries, three for each train. The natural language query describes functionality related to the voltage supply of the train. We use these queries because we know exactly what methods implement them, which enables us to use these methods as an oracle in the evaluation.

4.2 Research questions

The goal of our evaluation is to investigate the overall effectiveness of our approach. Moreover, we want to compare our FL+NL2RQ approach with a feature location approach (FL) that searches for relevant code directly from the NL query. Both FL+NL2RQ and FL perform the Code search step (see Figure 1) using the same LSI and SVD techniques. We have identified the following research questions:

RQ1: *How effective is our approach?*

This RQ evaluates the effectiveness of extracting relevant methods using the requirements query, which is obtained in our NL2RQ approach from the NL query. To answer this question, we run our approach taking as input the requirements and code of the train control product family and the NL query in order to determine whether the methods extracted are relevant.

RQ2: *Is the proposed FL+NL2RQ approach effective compared to the conventional FL approach?*

To answer this question, we compare our FL+NL2RQ approach (see right side of Figure 1) with the conventional FL approach (see left side of Figure 1). Although both approaches FL+NL2RQ and FL use LSI to code search, the difference between them is that we omit the transformation of the NL query into the requirements query in the implementation of the FL approach. Hence, the ranking of relevant code is obtained searching into the code of all products directly from the NL query.

4.3 Evaluation metrics

To answer the research questions, we follow the evaluation process that is depicted in Figure 3. To start with, a natural language query, the requirements and the code of the product family are taken as input for running our approach (FL+NLQ2RQ). Next, our approach generates a ranking of relevant methods for the query. In order to evaluate the results of our approach, we compare the results of the ranking with an oracle. The oracle is a table that holds the most relevant method for each query that we are going to use in the evaluation. We perform the code comparison by carrying out a diff because: 1) version control software has become really popular, 2) there is a wide amount of tool support that calculates differences between two source codes available, and 3) code comparison techniques have been used successfully for large scale systems [21, 16], therefore proving the computational cost of the operation to be affordable for large documents like ours.

Afterwards, we repeat the evaluation process for running the FL approach that searches directly into code. Finally, with the results of the comparisons, we are able to extract the evaluation metrics used to evaluate precision and recall. We use these metrics because they are the most common measures for the experiments within IR methods [9]. The recall and precision metrics are calculated as follows:

$$Recall = \frac{RankingElements \cap OracleElements}{OracleElements}$$

$$Precision = \frac{RankingElements \cap OracleElements}{RankingElements}$$

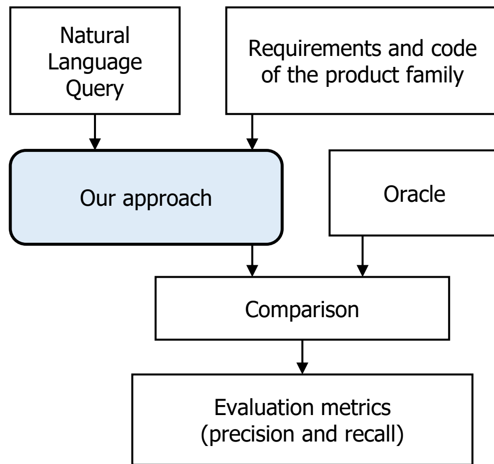

Figure 3: Evaluation process

Recall and Precision can be evaluated at a given cut-off rank as well, considering only the topmost results returned by the ranking. This measure is called recall or precision at n (Recall@n and Precision@n, respectively).

4.4 Tool support

The technological decisions for implementing the tool that supports our approach are the following:

- Open NLP[2] is a Natural Language Processing library developed by the Apache Software Foundation that is used for the POS tagging of the Keyword Extraction process of Step 1 (Requirement search).
- Snowball[3] is used for the stemming of the Keyword Extraction process of Step 1.
- Ecient Java Matrix Library (EJML)[4] is used for the implementation of the LSI and SVD of the textual analysis of requirements of Step 1 and the textual analysis of methods in the Code search. EJML is a basic linear algebra package for Java that provides an implementation of SVD.

For the evaluation, we use the DiffUtils library[5] to perform code diffs. The DiffUtils library is a Java open source library, which provides methods that we use to perform the necessary comparison between the relevant code obtained in our approach and the oracle.

4.5 Experimental results

RQ1: *The overall effectiveness of our approach*

We evaluate our approach by following the evaluation process described in Subsection 4.3 for the 21 different queries. For each query, our approach extracts the ranking of relevant code and calculates recall and precision values by comparing the methods of the ranking against the code of the oracle. The continuous line of Figure 4 shows the average values and the typical deviation for the recall and precision results obtained for the different queries and for the number of results in the ranking that are taken into account to compare the methods against the code of the oracle.

The recall results show that the average recall is 32.4% when the top 5 results are inspected (Recall@5) and it obtains 55.1% and 64.8% in terms of Recall@20 and Recall@40, respectively. Then, recall stabilizes.

With regard to the precision results, they show a precision score of 40% for the first returned result (Precision@1). Precision achieves 41.1% and 43.1% in terms of Precision@10 and Precision@20, it becomes stable when the top 33 are inspected. By looking at the typical deviation, it fluctuates as the recall and precision values change.

RQ2: *The effectiveness of FL+NLQ2RQ compared to FL*

To evaluate the effectiveness of FL+NLQ2RQ, we compare the FL+NLQ2RQ metrics with the FL metrics. The dashed line of Figure 4 shows the average values for the recall and precision results obtained for the different 21 queries in FL and their typical deviation. The recall results show that the average recall is 17.3% when the top 1 result is inspected (Recall@1) and it obtains 51.6% and 56.9% in terms of Recall@20 and Recall@40, respectively. Then, recall stabilizes when the top 76 results are inspected.

With regard to the precision results, they show a precision score of 22.8% for the first returned result (Precision@1), and it achieves 22.5% and 22% in terms of Precision@5 and Precision@10. Precision becomes stable when the top 72 are inspected. Regarding precision, it is possible to appreciate that FL+NLQ2RQ surpasses FL. The lowest value of precision in FL+NLQ2RQ is 37.9%, while the peak value of FL is 23.1%. The peak value of FL+NLQ2RQ tops in Precision@33 is 44.6%. FL+NLQ2RQ achieves an average improvement of 18.1% of precision against FL.

Moreover, it is important to note that FL+NLQ2RQ achieves its maximum recall and precision value before FL does. The peak recall value is obtained when 33 results are taken into account for FL+NLQ2RQ, while in FL it is not obtained until 76 results are reviewed. The peak precision value is obtained when 33 results are taken into account for FL+NLQ2RQ, while in FL it is not obtained until 67 results are observed. A better approach should allow developers to discover relevant methods for the query by examining fewer returned results in the ranking. Thus, the higher the metric values, the better the code search performance.

5. DISCUSSION

The product family, which is used as input in this experiment, was developed using Clone-and-Own (CAO) [10]. CAO is a common practice in software maintenance or evolution tasks that consists in copying the code and reusing it. In CAO, there are different relationships [3]:

- **Reimplemented.** Two methods do not share code between them, so their implementations are entirely different.
- **Modified.** Shared code exists between two methods.
- **Adapted.** One method includes all code from another method plus additional code.
- **Unaltered.** The code of two methods is strictly the same.

[2]http://opennlp.apache.org/
[3]http://snowball.tartarus.org/
[4]https://code.google.com/archive/p/efficient-java-matrix-library/
[5]https://code.google.com/archive/p/java-diff-utils/

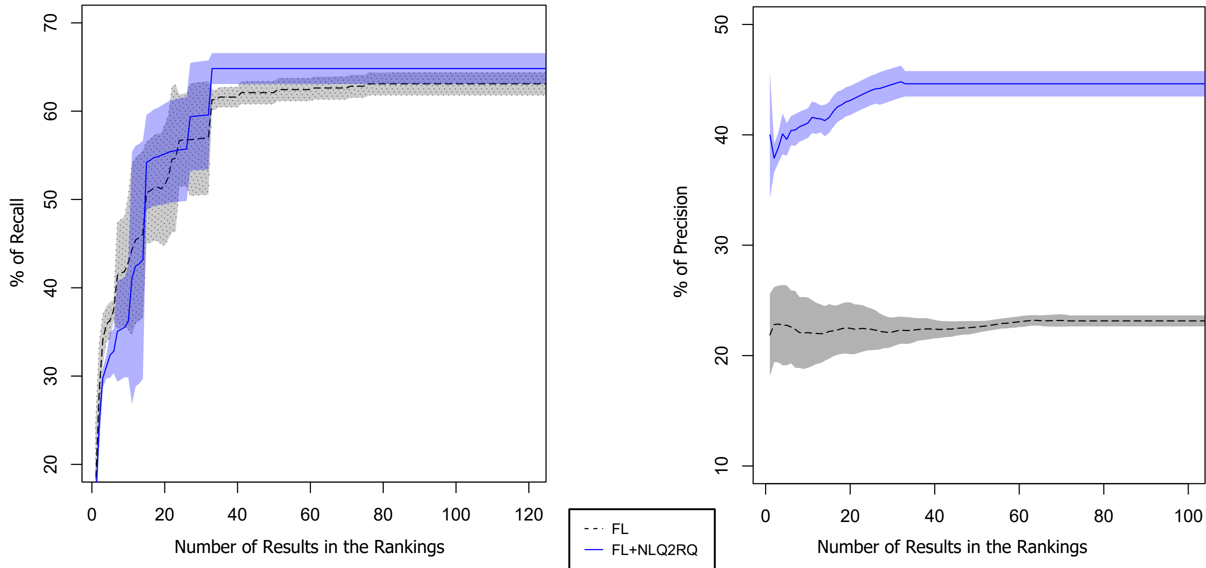

Figure 4: FL vs FL+NLQ2RQ: Recall and Precision results of the rankings

Through our evaluation, we have noticed that FL results were penalized when the methods that must be located are adapted or modified versions. It turns out that adapted or modified versions can have more terms apart from those that appear in the NL query. Figure 5 shows an example of this phenomenon. One of the terms that appears four times in the NL query after the post-stemming and LSI analysis is done is *COUPL*. The method that is the most relevant for this query according to the oracle is the *method 36 of Product 3* in which the *COUPL* term appears twice (see the left side of the figure).

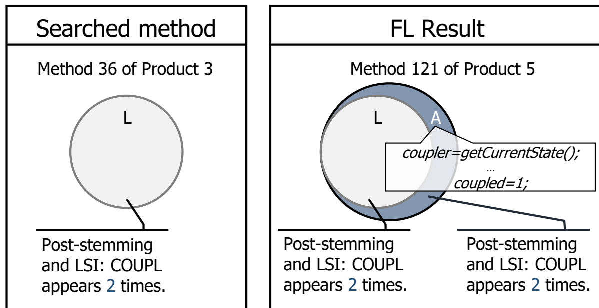

Figure 5: Example of penalized results using FL in adapted methods

Nevertheless, the result obtained using FL is the *method 121 of Product 5*, which is an adapted version of the searched method. The right side of Figure 5 shows that this adapted method has all the code that is present in method 36 of Product 3, referred as Legacy (L) in the figure, plus additional code, referred as Adapter (A) in the figure. This is because the adapter part of the method also contains terms that appear in the NL query (e.g., the sentences *coupler=getCurrentState()* and *coupled=1* make the *COUPL* term occurrences increase by two after the post-stemming and the LSI analysis are done). This increment makes the method more relevant for the NL query (e.g., the *COUPL* term in the adapted method appears four times as it does in the NL query, so the similarity measure improves). Hence, this adapted method is selected as result using FL (which penalizes precision) rather than selecting the searched method.

By contrast, the phenomenon of adapted or modified methods does not occur in requirements since they are rewritten and not adapted or modified. Therefore, our approach is not so sensitive to this phenomenon. For example, our approach eliminates Product 5 in the product scoping step since the similarity measure of its requirements excludes this product from being taken into account.

Although these results are preliminary and still require more experiments, it seems that our approach could provide better results in product families developed using CAO, so this work encourages further research in this direction.

6. THREATS TO VALIDITY

In this section we discuss some of the issues that might have affected the results of the evaluation. The first issue is related to the measures that are studied, and whether they represent what the researchers have in mind. To minimize this risk, we measured the factors of recall and precision, which are widely accepted [30, 29]. The second issue is the number of members in the family of trains. Although the number may look small, the products presented cover a wide range of railway types and they have been developed by two developer teams of our industrial partner.

The third issue is related to the generalization of the

findings. Software in the railway domain is representative of safety-critical systems like those present in the automotive domain or the aerospace domain. Nevertheless, our approach should be applied to other domains before assuring its generalization. Another issue is with regard to what extent the data and the analysis are dependent on the specific researcher. For our research, the data was recovered from trains chosen and provided by our industrial partner. The evaluation is performed by comparing the data with the oracle. Moreover, the final results are sensitive to the query that is used as input.

7. RELATED WORK

There are many feature location approaches proposed to find relevant code taking textual information as input [9]. For example, Marcus et al. [24] used IR techniques to map descriptions expressed in NL to source code. Cavalcanti et al. [6] used IR techniques to assign change requests in software maintenance or evolution tasks based on context information. Kimmig et al. [17] proposed an approach for translating NL queries to concrete parameters of the Eclipse JDT code query engine.

Recently, several approaches have been proposed to improve the effectiveness of feature location. For example, Wang et al. [33] proposed a code search approach, which incorporates user feedback to refine the query. Hill et al. [12] proposed to automatically extract NL phrases to categorize them into a hierarchy in order to help developers to discriminate the relevance of results and to reformulate queries. Zou et al.[35] investigated the 'answer style' of software questions with different interrogatives and proposed a re-ranking approach to refine search results.

Other approaches have been proposed to improve the effectiveness of feature location getting information from public repositories [11] or expanding a user query with semantic similar words from websites [32]. For example, Dietrich et al. [8] improved the efficacy of future queries using feedback captured from a validated set of queries and traceability links. Lv et al. [23] enrich each API with its online documentation to match the query based on text similarity.

Despite the fact that these approaches improve the effectiveness of feature location, they require an additional effort to enrich the code, to keep the documentation synchronized with the changes of the code throughout maintenance and evolution activities, and to incorporate users' feedback. Different from the above work, our work does not require additional efforts to enrich neither the code nor the query.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented our feature location approach (FL+NLQ2RQ) that transforms the NL query taken as input into a query that holds the most relevant requirements for the words used in the NL query. This level of indirection is the main innovation of our approach since it: 1) searches the most relevant requirements using keyword extraction (POS tags, stemming and stopwords) and textual analysis (LSI and SVD), 2) constructs the requirements query by joining the most relevant requirements, and 3) limits the scope of the code search space by selecting the code of the products that hold relevant requirements. Once the requirements query is obtained and the code search space is limited, relevant code is obtained.

Our evaluation has been performed in an industrial domain. Our industrial partner has provided the requirements and code of seven real-world trains. We have run our FL+NLQ2RQ approach to calculate the values of the most common measures (recall and precision) that are used in experiments with information retrieval methods. We have used these values to compare our approach with a conventional FL approach that searches directly into code.

Results have revealed that our approach is effective and outperforms the results obtained by searching directly in the code. The results show that our approach improves an average of 18.1% the results of precision against FL. Although these results are preliminary and still require more experiments, it seems that product families developed using CAO penalize the results of precision when the methods to be located are modified or adapted versions. In the context of product families developed with CAO, our approach provides better results since this behavior does not occur in requirements, so this work encourages further research in this direction.

In the future, we plan to evaluate differences of performance among different natural language processing techniques for building a single query from several requirements. Furthermore, we plan to further research the impact that the CAO relationships have in information retrieval for FL.

Acknowledgments

This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R).

9. REFERENCES

[1] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *12th International Software Product Line Conference*, pages 67–76, Sept 2008.

[2] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4, 2009.

[3] M. Ballarin, R. Lapeña, and C. Cetina. Leveraging feature location to extract the clone-and-own relationships of a family of software products. In *Proceedings of the 15th International Conference On Software Reuse*, ICSR '16, 2016.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 482–498, 1993.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[6] Y. a. C. Cavalcanti, I. d. C. Machado, P. A. d. M. S. Neto, E. S. de Almeida, and S. R. d. L. Meira. Combining rule-based and information retrieval techniques to assign software change requests. In

Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 325–330, New York, NY, USA, 2014. ACM.

[7] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[8] T. Dietrich, J. Cleland-Huang, and Y. Shin. Learning effective query transformations for enhanced requirements trace retrieval. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 586–591, Nov 2013.

[9] 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.

[10] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 25–34, 2013.

[11] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli. On-demand feature recommendations derived from mining public product descriptions. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 181–190, 2011.

[12] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 232–242, 2009.

[13] E. Hill, L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Automated Software Engineering (ASE)*, pages 524–527, 2011.

[14] W. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. String retrieval for multi-pattern queries. In *String Processing and Information Retrieval - 17th International Symposium, SPIRE*, pages 55–66, 2010.

[15] A. Hulth. Improved automatic keyword extraction given more linguistic knowledge. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 216–223, 2003.

[16] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[17] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 376–379, 2011.

[18] Krugle. http://www.krugle.com/.

[19] T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[20] C. Largeron, C. Moulin, and M. Géry. Entropy based feature selection for text categorization. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 924–928, 2011.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006.

[22] F. Liu, D. Pennell, F. Liu, and Y. Liu. Unsupervised approaches for automatic keyword extraction using meeting transcripts. In *Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the association for computational linguistics*, pages 620–628, 2009.

[23] F. Lv, H. Zhang, J. g. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on API understanding and extended boolean model. In *Automated Software Engineering (ASE2015)*, 2015.

[24] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004.

[25] M. Porter. Snowball: A language for stemming algorithms, Oct. 2001.

[26] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[27] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *ICPC*, pages 14–23, 2010.

[28] J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering*, pages 29–58. Springer, 2013.

[29] H. E. Salman, A. Seriai, and C. Dony. Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering. In *The 26th International Conference on Software Engineering and Knowledge Engineering*, pages 426–430, 2013.

[30] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[31] T. Strzalkowski, F. Lin, J. Perez-Carballo, and J. Wang. Building effective queries in natural language information retrieval. In *Proceedings of the Fifth Conference on ANLC*, pages 299–306, 1997.

[32] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 44–53, 2014.

[33] S. Wang, D. Lo, and L. Jiang. Active code search: Incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 677–682, 2014.

[34] J. Yang and L. Tan. Inferring semantically related words from software context. In *Mining Software Repositories (MSR)*, pages 161–170, 2012.

[35] Y. Zou, T. Ye, Y. Lu, J. Mylopoulos, and L. Zhang. Learning to rank for question-oriented software text retrieval. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, pages 1–11, 2015.