IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-6, Issue-11, Nov 2018 E-ISSN: 2347-2693

Bug Localization Approach on Lexical Pattern Extraction with Lexical Pattern Clustering

N. Kamaraj [1]*, A.V. Ramani [2]

[1]Information Technology, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Bharathiar University, Coimbatore
[2]Computer Science, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Bharathiar University, Coimbatore

Available online at: www.ijcseonline.org

Accepted: 21/Nov/2018, Published: 30/Nov/2018

Abstract— Bug localization is an important task of classification in software programming data set resources. Software programming data is used to find out the related programming codes, similar errors and files. Bug localization ranks the list of possible relevant entities. The bug localization task determines which source code entity is relevant to a particular bug report. In addition, the proposed paper also designs lexical pattern extraction clustering algorithm to classify the bugs in the given bugs report. It measures the semantic similarity between words which is an important component in various tasks on the web, such as relation extraction, community mining, and automatic extraction of metadata. To find out the various semantic relations existing between two given bug sentences, this paper proposes a new pattern extraction algorithm and a pattern clustering algorithm. The proposed method outperforms previously proposed web-based semantic similarity measures on the given data sets. It shows a high correlation with human ratings. Moreover, the above proposed method significantly progresses the accuracy in community mining task.

Keywords— Bug Localization, Lexical Pattern Extraction, Lexical Pattern Clustering, Information Retrieval

I. INTRODUCTION

The new research problem in software programming resource datasets are classification techniques which are applied for related software programming databases. The traditional manual bug localization is work-demanding since developers need to consider thousands and thousands of source code entities.

Current researches build bug localization classifiers which are based on information retrieval models or schema, to trace entities textually similar to bug reports.
Software developers typically utilize bug tracking databases, such as Bugzilla to manage upcoming bug reports in their software applications. For example, in the Eclipse project, developers receive approximately 115 new bug reports every day. Likewise, IBM Jazz projects get 105 new reports per day. Mozilla get 152 reports per day [1].

Developers need to spend considerable time as well as effort to study each new report and choose which source code entities are related for fixing bugs. Bug localization [2], [3] is a task which aims at classification problem: in which 'n' given source code entities and a bug report categorize, so that it belongs to one of 'n' entities. The classifier yields a ranked/sorted list of probable relevant entities, along with similarity percent for each entity in the given list. An individual entity is considered related if it certainly needs to be altered to resolve bug report; otherwise considered as irrelevant.

The recent bug localization researches make use of Information Retrieval(IR) classifiers to find out similarity between the given source code entities and the bug report. However, results may be indistinct and paradoxical: Some claim that Latent Dirichlet Allocation (LDA) [2] model is best, whereas others claim that Vector Space Model (VSM) provides the best performance [4], while still others argue that a new IR model is required [5]. Their conclusion is varying since they use different datasets, various performance measures, and different classifier parameters.

The classifier configurations characterize the value of all the configuration parameters that stipulate the behavior of a classifier, such that which source code entity is preprocessed, how its terms are weighted, and the similarity measure between source code entities and bug reports.

The rest of this paper is organized as follows: Section 2 reviews the existing bug localization approaches and explains related works and their limitations. Vector Space Model and Data preprocessing used in our study is briefed in Section 3. Our proposed bug localization aspects are detailed in Section 4. Section 5 discusses the lexical pattern extraction and clustering approaches for bug localization. Section 6 provides conclusion of the study.

II. LITERATURE

Existing IR-based bug localization classifiers consume. IR models to find out text-based similarities between source code entities (SC) (i.e., documents) and a bug report (BR) (i.e., query). For example, if a bug report (BR) contains the words, "Cannot open database required in login," then an IR model looks for entities that include ("database", "login"

etc.). If a bug report and entity contain many common words, then an IR-based classifier yields the entity a big relevancy score. Those classifiers contain many parameters which control their behavior [7].

Specifying a value for the given parameters fully characterizes the configuration of overall classifier. The parameters are common for all IR-based classifiers which govern how the input text data are presented and processed as follows:

a. Which parts of the source code should be taken, either comments, variables, or some other programming constructs, such as the previous bug reports coupled to each source code entity?

b. Which parts of the bug report should be taken, either title or description or both?

c. Should the source code and bug report be preprocessed? Do the variables/class names need to be split? Does the common stop need to be removed? Does the stemming need to be applied to their base form?

After configuration of these parameters, each IR model need to have own set of additional parameters which control reduction factors, similarity metrics, term weighting, and other metrics. The remainder of this section describes: the Vector Space Model, Latent Semantic Indexing and an enhancement to the Vector Space Model. Then various preprocessing steps are described that can be applied to bug reports and source code entities [15].

2.1 Vector Space Model

The Vector Space Model(VSM) algebraic model consists of the term-document matrix of a corpus [6]. The term-document matrix is an m x n matrix whose individual terms/words is represented as rows individual documents are represented as columns. The matrix's i^{th}, j^{th} entry is the weight of term wi in document dj. The term-document matrix is represented as column vectors in VSM; the vector contains the words weights present in document and zeros otherwise. Similarity between two documents is calculated based on comparison of two vectors. The two documents will be considered as similar, if they contain minimum one shared term; If they have more shared terms then their similarity score will be higher [14].

VSM uses the following parameters: Term weighting (TW): It is about the weight of the term/word in a
document. Values for the parameter are raw frequency, i.e., the number of occurrences of term/word in document or tf-idf (term frequency, inverse document frequency**)**. Similarity metric (Sim**Met**): It deals with the similarity between two document vectors. A popular parameter value is cosine distance or KL divergence [15].

2.2 Data Preprocessing

Several preprocessing steps are taken to reduce the noise and improve final resulting models. For example, Variable names are split using regular expressions into various parts based on naming conventions such as camel case (totalMark, under-scores (total_mark) and capitalization changes (TOTALMark).

Researchers have proposed advanced techniques to split identifiers based on automatic expansion and mining source code entities, which are more effective than basic regular expressions. Stemming of words are applied to detect each word's root (e.g., "studying" and "study" both become "study"), typically using the Porter algorithm.

The idea behind these optional steps is to confine developers' intention, which is guessed to be encoded within the variable or identifier names and comments in source code entities. The rest of the source code entities, i.e., language keywords, special syntax and stop words are viewed as irrelevant/noise and will not be advantageous as input for various IR models [8].

2.3 Current IR-Based Bug Localization approaches

At present, researchers have surveyed the use of IR models for bug localization approaches. Lukins et al. [7] evaluate the performance of LSI and LDA using small case studies. The authors built two IR classifiers on identifiers and comments of source code and compute similarity between each source code entity and a bug report using cosine similarity and conditional probability similarity metrics. After performing case studies on Mozilla and Eclipse (with a total of five and three bug reports, respectively), the authors found that LDA often outperforms LSI well. It is noted that authors use manual query expansion which influenced their results.

Nguyen et al. [5] introduced a latest topic model based on LDA, termed as Bug Scout, in an effort to progress bug localization performance. Bug Scout explicitly reflected on past bug reports, in addition to comments and identifiers, on behalf of source code documents, using two data sources concurrently to recognize key technical concepts. The authors applied BugScout to four dissimilar projects and found that BugScout enhanced performance by up to 20 per cent over LDA which is applied only to source codes.

2.4 Drawbacks of current research

Researchers consider only a few configurations of classifiers (see Table 1), often with no justification given for why parameter values were chosen out of large space of possible values in current research. Worse, many parameter values were left unspecified, making replication of their results hard or impractical.

Given that there were a number of choices for each parameter in configurations, and the parameters were independent, there were thousands of possible configurations for each underlying IR model.

The effectiveness of each configuration which parameters are important and works best are currently unknown. As a result, researchers and practitioners are left to guess which configurations to use in their project.

Researchers have combined various approaches to do concept location. Poshyvanyk et al. [8] combine LSI with a dynamic feature location approach named scenario-based probabilistic ranking. These two approaches operate on various datasets and use various analysis methods. The results of combined approach are better than individual approach as evidenced by two different case studies on large projects. Poshyvanyk and Marcus [9] combine LSI and Formal Concept Analysis to accomplish similar effects. Cleary et al. [10] unite quite a few IR models with Natural Language Processing techniques and conclude that NLP techniques do not get better results. Finally, Revile [11] combine LSI, web mining algorithms and dynamic analysis for feature location. They find that combination outperforms any of the individual approaches.

III. METHODOLOGIES

The goal of this case study is to assess the space of bug localization classifier configurations: which preprocessing steps, data representations, and other IR model parameters result in best bug localization routine.

3.1 Case Study Design

The design of the case study is outlined: which classifiers to be defined, which software projects need to be tested, the data collection technique, and the performance metrics, i.e., criterion functions need to be used.

3.2. Defined Classifiers

Two families of classifiers are considered: IR-based classifiers and entity metric-based (EM-based) classifiers. Tables 1 and 2 list the parameters and values in the classifier assessment, for IR-based classifiers and entity metric-based classifiers, respectively. It aims to choose realistic values that are representative of those used most often in the study, while keeping a number of configurations. Each parameter and its possible values.

IR-based classifiers are built based on three popular IR models: VSM, LSI, and LDA. For each IR model, it must be decided which bug report representation is to be used for query, which source code entity representation is to be used to build index, how to preprocess bug report and source code representation, and remaining parameter values for particular IR model. For source code entity representation, six values are considered. First three are based on text of the source code entity itself: only identifier names, i.e., method names and variables (B1), only comments (B2), and identifiers as well as comments (B3). Like Nguyen et al. [6], the past bug reports (PBR) related to a particular source code entity are related. To do so, it is corresponded to the source code entity as a collection of text of all of its PBRs. So, a new bug report may be more textually similar to an old bug report than to comment or identifier names of an entity, giving the IR model a improved chance for success. They consider two values are: using the entire PBRs of an entity (B4) and using just ten most recent, i.e., min(10, |PBR|) PBRs of a given entity (B5). Finally, it is regarded as all possible data for an entity: its comments, identifier and all PBRs (B6). For bug report, i.e., query representation, three values are deemed: title of the bug report only (A1), description of bug report only (A2), and title as well as and description of bug report (A3). In this study, the comments or other metadata related to the bug report are not considered as this information is usually not available at time of bug localization.

No query expansion technique carried out besides preprocessing steps described below. There are three common preprocessing steps: removing stop words, splitting identifiers using basic simple regular expressions and apply stemming. The programming language keywords and punctuation can also be eliminated. Since the application of each preprocessing step is twofold, i.e., performed/not performed, three preprocessing steps can be applied independently. A total of eight possible preprocessing techniques (C0-C7) are tested.

The VSM model has two parameters: term weighting (A) and similarity score (B). For term weighting (A), the tf-idf (D1), sub linear tf-idf (D2) weighting scheme as well as more basic Boolean (D3) weighting scheme are considered.

For similarity score (B), both cosine (E1) and overlap similarity (E2) scores are considered. The LSI model has three parameters: similarity score, term weighting and number of topics. The exact three terms weighting schemes are considered, it is done as for the VSM model (F1-F3).
The similarity score constant at cosine (H1) is embraced, since research has shown this is the best similarity score for LSI. Finally, it is deemed that four values for number of topics: 32, 64, 128, and 256 (G32-G256). Smaller values yield coarser-grained topics, while larger values yield finer-grained topics. At present, there is no automatic methodology for selecting an optimal number of topics and so the values that cover the typical value ranges are selected.

Table 1. The IR Family of Classifiers Studied

Parameter	Value
Parameters common to the entire IR classifiers	
Preprocessing Steps	C0 (None) C1 (Split only) C2 (Stop only) C3 (Stem only) C4 (Split+stop) C5 (Split+stem) C6 (Stop+stem) C7 (Split+stop+stem)
Entity Representation	B1 (Variables only) B2 (Comments only) B3(Variables+comments) B4 (OBR-All) B5 (OBR-10 only) B6(Variables+comments+OBR-All
Bug report	A1 (Title only)

representation	A2 (Description only) A3 (Title+description)
Parameters for VSM only	
Term weight	D1(tf-idf) D2 (Sublinear tf-idf) D3 (Boolean)
Similarity metric	E1 (Cosine) E2 (Overlap)
Parameters for LSI only	
Term weight	F1 (tf-idf) F2 (Sublinear tf-idf) F3 (Boolean)
Number of topics	G32 (32topics) G64 (64topics)
Similarity metric	H1 (Cosine)
Parameters for LDA only	
Number of iterations	I1 (Unit model convergence)
Number of Topics	J32 (32topics) J64 (64topics)
	K1 (Optimized based on K)
	L1 (Optimized based on K)
Similarity	N1 (Conditional probability)

Configuration parameters and values are shown that they are considered for each of the three IR models: VSM, LSI and LDA. OBR is Old Bug Reports.

The LDA model has five parameters: a) number of topics, b) a document-topic smoothing parameter, c) a topic-word smoothing parameter, d) number of sampling iterations, and e) similarity score. They considered two values for the number of topics are: 32 and 64 (J32-J64), to be consistent with the choices for LSI model. The LDA implementation is called MALLET [12], automatically optimizes for the document-topic and topic-word smoothing parameters, so the values for these parameters are not set manually. It is also not specified that the number of iterations manually, and let the model run till convergence.

Finally, conditional probability score is considered (N1), as it is most related for IR applications. Conditional probability does not require the bug reports to be included in the LDA model at runtime. In contrast, other similarity measures are not practical as they require LDA to be retuned (the bug reports and the source code entities) every time a new bug report appears.

EM (Entity Metric) based Classifiers: The last decade was very active on research in the area of bug prediction. This research focuses at measuring features of source code, such as lines of code (LOC), change proneness, past bug-proneness and logical coupling between classes, to forecast which source code entities contain bugs/errors.

The researcher is the first to put forward importing these methods from bug prediction studies to generate bug localization classifiers. To this end, EM-based classifiers first calculate one or more metrics on source code entities. Then, the classifiers rank the source code entities based on metrics. For example, a higher LOC metric indicates more errors, so one EM-based classifier would order entities by their LOC. It is noted that dissimilar to other IR-based classifiers.

The rankings of EM-based classifiers are not based on given bug reports and so the same ranked list will be generated for each and every bug report. Still, it is noted (and the error prediction literature confirmed) that since errors are highly concentrated in small number of source code entities, this list is likely to be precise for given bug report.

The EM-based classifier has a single parameter: which entity metric is used to find out the bug-proneness of an entity. Four metrics are considered: LOC of an entity, churn of the entity (number of LOC that were added, changed or deleted since the preceding version), cumulative bug count of the entity (the number of bugs that have been allied with this entity earlier), new bug count of the entity (the number of bugs only since the earlier version). Previous research shows that these metrics are fine predictors of bug-proneness of the entity, so it is supposed that the metrics to have reasonable performance for bug localization.

Fully Factorial Design: To quantify the performance of all possible classifiers (given the considered parameters and their possible values), a fully factorial design of the case study is used. In this design, every possible combination of parameter values is explored. Configuration parameters and values are shown that we consider.

Table 2. The EM Family of Classifiers Studied

Parameter	Value
Metric	M1 (Lines of Code) M2 (Churn) M3 (New error count) M4 (cumulative error count)

IV. ANALYSIS AND MPLEMENTATION

4.1 Co-occurrence measures

Generally, page counts-based co-occurrence measures do not consider the local perspective in which source code/bug co-occurs. This can be problematic if one or both words are different but with the same meaning, or when page counts are unreliable. On the other hand, the snippets present in the source code for the conjunctive query of two words provide useful clues related to the semantic relations that exist between two words. A snippet contains a window of text selected from a document (source code entity or bug report) that includes the queried words. Snippets are useful for

finding similarity because, most of the time, a user can read that and make a decision whether a particular content is relevant. The circumstance is also computationally competent. For example, consider the snippet

Relation is a synonym word for Table in RDBMS terminology.

A snippet with synonym

Here, the phrase 'is a' indicates a semantic relationship between 'Table' and 'Relation'. Many such phrases point out semantic relationships. For example, 'also known as', 'is a', 'part of', 'is an example of' all indicate semantic relations of different types. In the example given above, words indicating the semantic relation between 'Table' and 'Relation' come into view between the term words. Replacing the term words by variables X and Y, it is able to form the pattern 'X is a Y' from the example given above.

In spite of the efficiency of using snippets, pose two major challenges are posed: first, a snippet could be a fragmented sentence; second, a snippet might be produced by selecting multiple text fragments from different portions in a document (source code entity or bug report). Because most syntactic parsers assume complete sentences as input, deep parsing of snippets yields incorrect results. So, it is proposed that a lexical pattern extraction algorithm using snippets, to distinguish the semantic relations that exist between two term words.

4.2 Lexical Pattern Extraction and Clustering

The similarity between source code entities and bug reports present lexical pattern extraction and clustering approach which better outperforms in finding similarity.

Input: Patterns A (a_1, … ,a_n), threshold θ
Output: Clusters C

SORT (A)
C ← {}
for pattern ai ∈ A do
max ← - ∞
c^* ← null
for cluster cj ∈ C do
sim ← cosine (ai, cj)
if sim > max then
max ← sim
c ← cj
end if
end for
if max > θ then
c* ← c* ⊕ ai
else
C ← C ∪ {ai}
end if
end for
return C

4.2.1 Lexical Patterns Extractions

These patterns have been used in various natural language processing tasks such as extracting meronyms or hypernyms, paraphrase extraction and question answering. Although we might produce a snippet by selecting multiple text fragments from various portions in a source code entity or bug report, a predefined delimiter can be used to break up different fragments. For example, in Google search engine, the delimiter "..." is used to break up various fragments in a snippet. Such delimiters are used to split a snippet before executing the proposed lexical pattern extraction algorithm on each fragment.

Given two term words P and Q, one can search in the source code entity using the wildcard query "P * * * * * Q" and get snippets. The "*"operator matches one word or none in a source code entity. Therefore, the wildcard query obtains snippets in which P and Q appear within a window of seven words. Here it is assumed that the seven word window is sufficient to cover most relations between two words in source code blocks. In fact, most of the lexical patterns extracted by the proposed method contain less than five words. One can try to approximate the local meaning of two words using wildcard queries. For example, a snippet retrieved for the query "Cannot login."

cannot open database required in login

A snippet retrieved for pattern 'cannot * * * login'.

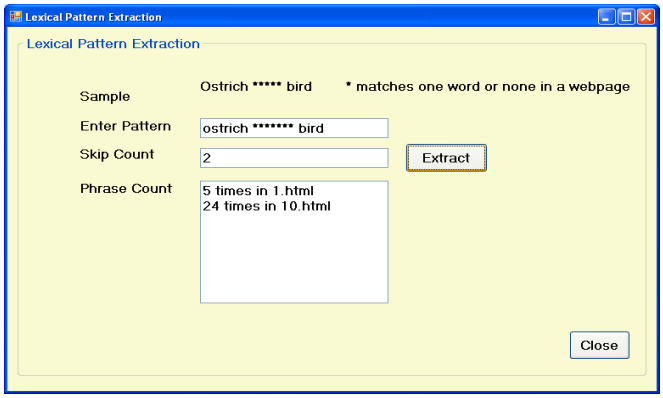
For a snippet, retrieved for a word pair (P, Q), the two terms P and Q are replaced respectively, with variables X and Y. Numeric values are replaced by D, an indicator for digits. Then it is preparing that all subsequences of words from snippet that satisfy all of the following conditions:

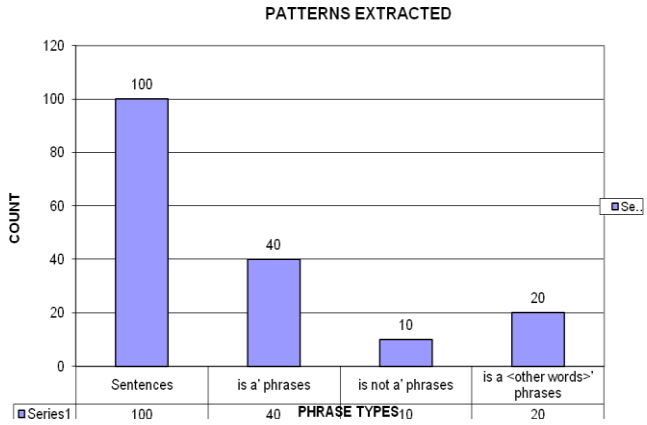
a. A subsequence should contain one occurrence of each X and Y.
b. The maximum length of a subsequence is 'L' words.
c. A subsequence is allowed to leave out one or more words. However, more than 'g' numbers of words are not left consecutively. Moreover, the total number of words skipped in a subsequence should not exceed 'G'.
d. While generating subsequences word is not left out, For example, this condition ensures that from the snippet X is not a Y, subsequence X is a Y is not produced. Finally, the frequency of generated subsequences is counted and subsequences are used that occur more than T times as lexical patterns.

The sample training data which contains paragraphs with sentences containing lexical pattern phrases like 'is a', 'is a flying', 'is a flying bird' and the like. For example, if a sentence is 'Ostrich is a flying bird' and the other sentence is 'Ostrich is an Australian flying bird' then during the lexical

pattern extraction (with skip count set to 1 or 2) The sentence 'Ostrich * * * * bird' is given as into input. Then both the phrases 'is a flying' and 'is an Australian flying' are extracted out.

Lexical Patterns Extractions





4.2.2 Lexical Pattern Clustering

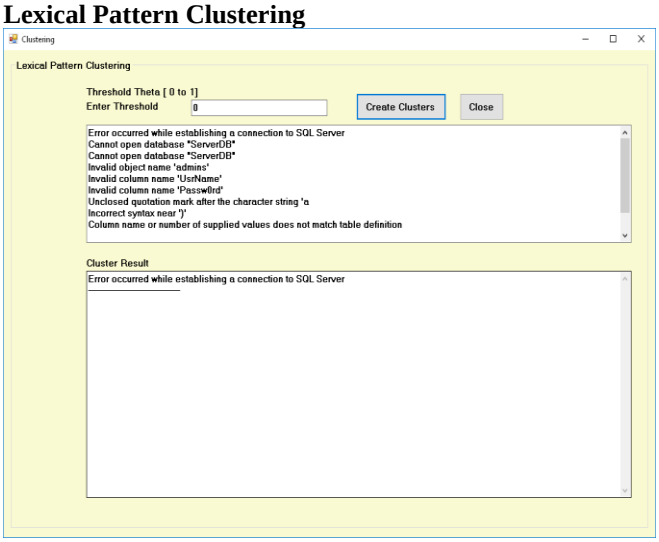
Normally, a semantic relation can be articulated using more than one pattern. Consider the two distinct patterns, X is a Y, and X is a large Y. Both patterns point out that there exists an is-a relation between X and Y. Different patterns identification that express the same semantic relation allows to signify the relation between two term words precisely. According to the distributional hypothesis [13], term words that present in the same perspective have similar meanings. Distributional hypothesis has been used in different related processes, like identifying related words and digging out paraphrases. If the term word pairs is considered that assure (i.e., co-occur with) a given lexical pattern as the context of that lexical pair, then from the distributional hypothesis, it follows that the lexical patterns which are similarly distributed over word pairs must be semantically similar.

The patterns are clustered using the lexical pattern clustering algorithm. The patterns are clustered and then the count and co-occurrence of the word can be considered. Based on this the word can be extracted. The cluster can be grouped based on the threshold value, the words are clustered and then the results are produced. It is a brief window of text extracted by a search engine around the query term in a document. It provides useful information regarding the local context of the query term. Snippets, a brief window of text extracted by a search engine around the query term in a document, provide useful information regarding the local context of the query term. Semantic similarity measures defined over snippets, have been used in query expansion, personal name disambiguation, and community mining.

Processing snippets is also efficient because it obviates the trouble of downloading web pages, which mining is time consuming depending on the size of the pages. However, a widely acknowledged drawback of using snippets is that, because of the huge scale of the web and the large number of documents in the result set, only those snippets for the top ranking results for a query can be processed efficiently.

Ranking of search results, hence snippets, is determined by a complex combination of various factors unique to the underlying search engine. Therefore, no guarantee exists that all the information requered to measure semantic similarity between a given pair of words is contained in the top - ranking snippets. Drawback because of the huge scale of the web and the large number of documents in the results set, only those snippets for the top ranking results for a query can be processed efficiently.

Lexical Pattern Clustering



V. CONCLUSION

Resolving the bug localization problem has major implications for programmers or developers because it can reduce the 'time and effort' required to maintain/manage software. In this paper, we detail the bug localization problem as one of classification and analyze the effect classifier configuration on bug localization performance, in addition with, whether classifier combination could help. The summary of our main findings are as follows:

a) The individual IR-based classifier uses the Vector Space Model, with the index built using tf-idf term weighting on available data in source code entities (i.e., variables, comments, and old bug reports for all entities), which has been stemmed, stopped, and split, and queried with all available data in bug report (i.e., title and description) with cosine similarity.

b) Classifier combination helps in almost all cases, no matter the underlying classifiers used or the specific combination technique used.

We also like to investigate further whether preprocessing bug reports by removing noise in the code snippets could be advantageous to bug localization results. We also found a semantic similarity measure using both page counts and snippets retrieved from code snippets. We developed a lexical pattern extraction method to extract numerous semantic relations exist between two term words. In addition, a sequential pattern clustering algorithm was developed to identify various lexical patterns that describe the identical semantic relation. Lexical pattern clusters were used to define features for a word pair.

REFERENCES

[1] Mozilla Foundation, Bugzilla.2012.

[2] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn, Bug Localization Using Latent Dirichlet Allocation, Information and Software Technology, vol. 52, no. 9,pp. 972-990, 2010.

[3] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi,H.V. Nguyen, and T.N Nguyen, "A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report," Proc. 26th Int'l Conf Automated Software Eng., pp. 263-272, 2011.

[4] S. Rao and A. Kak, "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models," Proc. Eighth Working Conf. Mining Software Repositories, pp. 43-52, 2011.

[5] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi, H.V. Nguyen, and T.N. Nguyen, "A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report," Proc. 26th Int'l Conf. Automated Software Eng., pp. 263-272, 2011.

[6] G. Salton, A. Wong, and C.S. Yang, "A Vector Space Model for Automatic Indexing," Comm. ACM, vol. 18, no. 11, pp.613-620, 1975.

[7] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn, Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation, Proc. 15th Working Conf. Reverse Eng., pp. 155-164, 2008.

[8] D.Poshyvanyk, Y. Gueheneuc, A. Marcus, G.Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," IEEE Trans. Software Eng., vol. 33, no. 6 pp. 420-432, June 2007.

[9] D.Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," Proc. 15th Int'l Conf. Program Comprehension, pp. 37-48, 2007.

[10] B.Cleary, C.Exton, J.Buckley, and M.English, "An Empirical Analysis of Information Retrieval Based Concept Location Techniques in Software Comprehension," Empirical Software Eng., vol. 14, no. 1, pp. 93-130, 2008.

[11] M. Revelle, B. Dit, and D. Poshyvanyk, Using Data Fusion and Web Mining to Support Feature Location in Software, Proc. 18th Int'l Conf. Program Comprehension, pp. 14-23, 2010.

[12] A.K. McCallum, "Mallet: A Machine Learning for Language Toolkit," http://mallet.cs.umass.edu, 2002.

[13] Z. Harris, "Distributional Structure," Word, vol. 10, pp. 146-162, 1954.

[14] J.Ren,M.Harman,M.Di Penta," Cooperative Co-evolutionary optimization of software project staff assignments " in Proc.Int.Symp.Search –Based Sotw. Eng., 2011 , pp. 127-141

[15] C.D.Manning, P.Raghavan and H.Schutze, Introduction to Information retrieval, vol.1, Cambridge Univ. Press Cambridge, 2008.

Authors Profile

N.Kamaraj Assistant Professor in Information Technology Department at SRMV College of Arts and Science, Coimbatore, TamilNadu. He has many Publications in national & international journal and Conferences. Currently he is doing research in the area of Software Engineering

Dr.A.V.Ramani Completed his PhD degree fromBharathiya university, TamilNadu, India. He has 25 publications to his credit He has 30 years of teaching experience .Presently he is working as a associate Professor and HOD in computer Science, SRMVCAS, Coimbatore, Tamilnadu, India

