LEARNING TO RANK AND CLASSIFICATION OF BUG REPORTS USING SVM AND FEATURE EVALUATION

[1]S.Rajeswari, [2]S. Sharavanan, [3]R.Vijai and [4]RM. Balajee

[1]PG Scholar/ Department of CSE,

[2]Professor & Head / Department of CSE,

[3, 4] AP / Department of CSE,

Annapoorana Engineering College, Salem.

Email: rajeswariselvaraj89@gmail.com

Submitted: May 27, 2017 ***Accepted: June 15, 2017*** ***Published: Sep 1, 2017***

Abstract- When a new bug report is received, developers usually need to reproduce the bug and perform code reviews to find the cause, a process that can be tedious and time consuming. A tool for ranking all the source files with respect to how likely they are to contain the cause of the bug would enable developers to narrow down their search and improve productivity. This project introduces an adaptive ranking approach that leverages project knowledge through functional decomposition of source code, API descriptions of library components, the bug-fixing history, the code change history, and the file dependency graph. Given a bug report, the ranking score of each source file is computed as a weighted combination of an array of features, where the weights are trained automatically on previously solved bug reports using a learning-to-rank technique. I applied SVM (Support Virtual Machine) to classify the bug reports to identify, which category the bug belongs to. It helps to fix the critical defects early. The ranking system evaluated on six large scale open source Java projects, using the before-fix version of the project for every bug report. The experimental results show that the learning-to-rank approach outperforms three recent state-of-the-art methods. In particular, proposed method makes correct recommendations within the top 10 ranked source files for over 70 percent of the bug reports in the Eclipse Platform and Tomcat projects.

Index terms*:* Learning to rank, SVM, Preprocessing, CF(collaborative Filtering)

.

I. INTRODUCTION

A software bug or defect is a coding mistake that may cause an unintended or unexpected behavior of the software component. Upon discovering an abnormal behavior of the software project, a developer or a user will report it in a document, called a bug report or issue report. A bug report provides information that could help in fixing a bug, with the overall aim of improving the software quality. A large number of bug reports could be opened during the development life-cycle of a software product. A developer who is assigned a bug report usually needs to reproduce the abnormal behaviour and perform code reviews in order to find the cause. If the bug report is constructed as a query and the source code files in the software repository are viewed as a collection of documents, then the problem of finding source files that are relevant for a given bug report can be modelled as a standard task in information retrieval(IR).

The ranking function is defined as a weighted combination of features, where the features draw heavily on knowledge specific to the software engineering domain in order to measure relevant relationships between the bug report and the source code file. While a bug report may share textual tokens with its relevant source files, in general there is a significant inherent mismatch between the natural language employed in the bug report and the programming language used in the code. Ranking methods that are based on simple lexical matching scores have sub optimal performance, in part due to lexical mismatches between natural language statements in bug reports and technical terms in software systems. The system contains features that bridge the corresponding lexical gap by using project specific API documentation to connect natural language terms in the bug report with programming language constructs in the code.

II. LITERATURE SURVEY

Hal Daume III and Daniel Marcu [1] Entity detection and tracking (EDT) is the task of identifying textual mentions of real-world entities in documents, extending the named entity detection and co reference resolution task by considering mentions other than names (pronouns, definite descriptions, etc.). Like NE tagging and co reference resolution, most solutions to the EDT task separate out the mention detection aspect from the co reference aspect. By doing so, these solutions are limited to using only local features for learning. In contrast, by modeling both

aspects of the EDT task simultaneously, we are able to learn using highly complex, non-local features. Develop a new joint EDT model and explore the utility of many features, demonstrating their effectiveness on this task. In many natural language applications, such as automatic document summarization, machine translation, question answering and information retrieval, it is advantageous to pre-process text documents to identify references to entities. An entity, loosely defined, is a person, location, organization or geopolitical entity (GPE) that exists in the real world. Being able to identify references to real-world entities of these types is an important and difficult natural language processing problem. It involves finding text spans that correspond to an entity, identifying what *type of entity* it is (person, location, etc.), identifying what *type of mention* it is (name, nominal, pronoun, etc.) and identifying which *other* mentions in the document it co refers with. The difficulty lies in the fact that there are often many ambiguous ways to refer to the same entity.

RiponK.Saha, MathewLease, Dewayne E.perry [2] Locating bugs is important, difficult, and expensive, particularly for large-scale systems. To address this, natural language information retrieval techniques are increasingly being used to suggest potential faulty source files given bug reports. While these techniques are very scalable, in practice their effectiveness remains low in accurately localizing bugs to a small number of files. Key insight is that structured information retrieval based on code constructs, such as class and method names, enables more accurate bug localization. BLUiR, which embodies this insight, requires only the source code and bug reports, and takes advantage of bug similarity data if available. Build BLUiR on a proven, open source IR toolkit that anyone can use. The research work provides a thorough grounding of IR-based bug localization research in fundamental IR theoretical and empirical knowledge and practice. Evaluate BLUiR on four open source projects with approximately 3,400 bugs. Results show that BLUiR matches or outperforms a current state-of-the art tool across applications considered, even when BLUiR does not use bug similarity data used by the other tool.

Giuliano antonial, Yann-gael Gueheneuc [3] Feature identification is a well-known technique to identify subsets of a program source code activated when exercising a functionality. Several approaches have been proposed to identify features. An approach to feature identification and comparison for large object-oriented multi-threaded programs using both static and dynamic data. Using processor emulation, knowledge filtering, and probabilistic ranking to overcome the

difficulties of collecting dynamic data, i.e., imprecision and noise. We use model transformations to compare and to visualise identified features. Compare the new approach with a naive approach and a concept analysis-based approach using a case study on a real-life large object-oriented multi-threaded program, Mozilla, to show the advantages of our approach. Maintenance of legacy software involves costly and tedious program understanding tasks to identify and to understand data structures, functions, methods, objects, and classes and, more generally, any high-level abstractions required by maintainers to perform their tasks. Source code browsing is the most common activity performed during software maintenance because obsolete (or missing) documentation forces maintainers to rely on source code only. Unfortunately, source code browsing becomes very resource consuming as the size and the complexity of programs increase. An alternative to source code browsing is automated design recovery. Central to design recovery is the recovery of *higher-level abstractions beyond those obtained by examining the system itself*. We propose an approach to support the recovery of higher-level abstractions through program feature identification and comparison.

Sushil K Bajracharya, Joel ossher, cristina V Lopes[4] Developers often learn to use APIs (Application Programming Interfaces) by looking at existing examples of API usage. Code repositories contain many instances of such usage of APIs. However, conventional information retrieval techniques fail to perform well in retrieving API usage examples from code repositories. This paper presents Structural Semantic Indexing (SSI), a technique to associate words to source code entities based on similarities of API usage. The heuristic behind this technique is that entities (classes, methods, etc.) that show similar uses of APIs are semantically related because they do similar things. We evaluate the effectiveness of SSI in code retrieval by comparing three SSI based retrieval schemes with two conventional baseline schemes. The results of the evaluation show that SSI is effective in improving the retrieval of examples in code repositories. The large availability of software on the Web is having a fundamental impact on software development in at least two ways. First, web search engines have enabled the retrieval of software that would otherwise be undiscoverable. Second, thanks to the wide availability of all sorts of libraries, developers often prefer to reuse components than to write something from scratch. nowadays, it is possible to find libraries that implement virtually every well-known piece

of functionality – both because those libraries exist and because they are findable via search engines.

Martin Buger, Andreas Zeller[5] A program fails. Taking a single failing run, we record and minimize the interaction between objects to the set of calls relevant for the failure. The result is a minimal unit test that faithfully reproduces the failure at will: "Out of these 14,628 calls, only 2 are required". In a study of 17 real-life bugs, our JINSI prototype reduced the search space to 13.7% of the dynamic slice or 0.22% of the source code, with only 1–12 calls left to examine. When a program fails, a developer must debug it in order to fix the problem. Debugging consists of two essential steps. The first is reproducing the failure. Reproducing is essential because without being able to reproduce the failure, the developer will have trouble diagnosing the problem and eventually demonstrating that it has been fixed. Reproducing failures depends on the knowledge about the circumstances that lead to a failure; if these are little known or hard to recreate, reproducing can be a tough challenge. The second step in debugging is finding the defect. For this purpose, one must trace back the cause-effect chain that leads from defect to failure—a search across the program state and the program execution to identify the cause of the problem.

Nicholas Bettenburg, sascha just[6] CUEZILLA prototype is such a tool and measures the quality of new bug reports; it also recommends which elements should be added to improve the quality. We trained CUEZILLA on a sample of 289 bug reports, rated by developers as part of the survey. In our experiments, CUEZILLA was able to predict the quality of 31–48% of bug reports accurately. Bug reports are vital for any software development. They allow users to inform developers of the problems encountered while using software. Bug reports typically contain a detailed description of a failure and occasionally hint at the location of the fault in the code (in form of patches or stack traces). However, bug reports vary in their quality of content; they often provide inadequate or incorrect information.

III. EXISTING SYSTEM

A developer who is assigned a bug report usually needs to reproduce the abnormal behaviour and perform code reviews in order to find the cause. However, the diversity and uneven quality of

bug reports can make this process nontrivial. Essential information is often missing from a bug report. Bacchelli and Bird surveyed 165 managers and 873 programmers, and reported that finding defects requires a high level understanding of the code and familiarity with the relevant source code files. In the survey, 798 respondents answered that it takes time to review unfamiliar files. While the number of source files in a project is usually large, the number of files that contain the bug is usually very small. Therefore, we believe that an automatic approach that ranked the source files with respect to their relevance for the bug report could speed up the bug finding process by narrowing the search to a smaller number of possibly unfamiliar files. If the bug report is construed as a query and the source code files in the software repository are viewed as a collection of documents, then the problem of finding source files that are relevant for a given bug report can be modeled as a standard task in information retrieval (IR). As such, we propose to approach it as a ranking problem, in which the source files (documents) are ranked with respect to their relevance to a given bug report (query). In this context, relevance is equated with the likelihood that a particular source file contains the cause of the bug described in the bug report. The ranking function is defined as a weighted combination of features, where the features draw heavily on knowledge specific to the software engineering domain in order to measure relevant relationships between the bug report and the source code file. While a bug report may share textual tokens with its relevant source files, in general there is a significant inherent mismatch between the natural language employed in the bug report and the programming language used in the code.

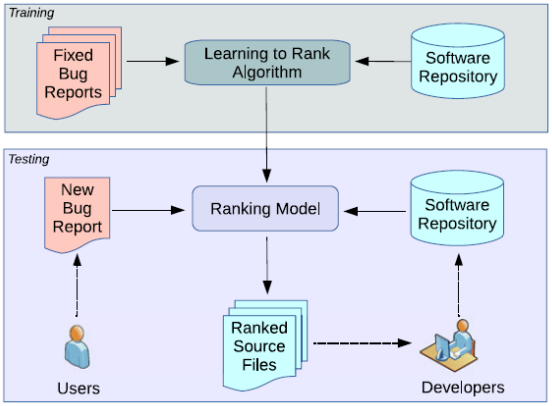


Fig.1 Existing System Architecture

IV. PROPOSED SYSTEM

To locate a bug, developers use not only the content of the bug report but also domain knowledge relevant to the software project. We introduced a learning-to-rank approach that emulates the bug finding process employed by developers. The ranking model characterizes useful relationships between a bug report and source code files by leveraging domain knowledge, such as API specifications, the syntactic structure of code, or issue tracking data. Experimental evaluations on six Java projects show that our approach can locate the relevant files within the top 10 recommendations for over 70 percent of the bug reports in Eclipse Platform and Tomcat. Furthermore, the proposed ranking model outperforms three recent state-of-the-art approaches. Feature evaluation experiments employing greedy backward feature elimination demonstrate that all features are useful. When coupled with runtime analysis, the feature evaluation results can be utilized to select a subset of features in order to achieve a target trade-off between system accuracy and runtime complexity.

The proposed adaptive ranking approach is generally applicable to software projects for which there exists a sufficient amount of project specific knowledge, such as a comprehensive API documentation and an initial number of previously fixed bug reports. Furthermore, the ranking performance can benefit from informative bug reports and well documented code leading to a better lexical similarity, and also have a plan to use the ranking SVM with nonlinear classification to classify the source code files in given seven types.

ARCHITECTURE DIAGRAM:

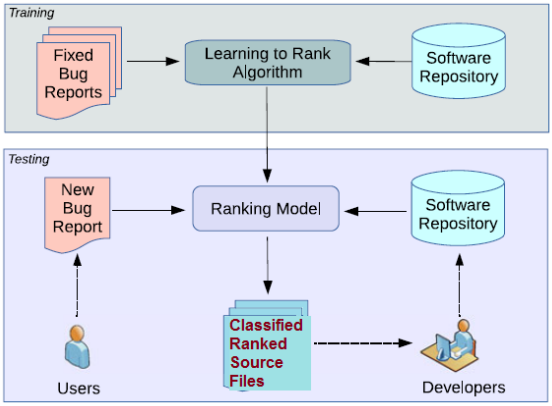


Fig.2 System Architecture

4.1 MODULES

1. Pre-processing
2. Collaborative Filtering
3. Feature Selection
4. Ranking
5. Classification result

4.1.1 Pre-processing

The first step towards handling and analyzing textual data formats in general is to consider the text based information available in free formatted text documents or text. Initially the pre-processing is done by the following processes

4.1.1.1 Removal of Stop Words

The first step is to remove the un-necessary information available in the sentence of stop words. These include some verbs, conjunctions, disjunctions and pronouns, etc. (e.g. is, am, the, of, an, we, our)

4.1.1.2 Removal of Stem Words

Stemming words e.g. ‘deliver’, ‘delivering’ and ‘delivered’ are stemmed to ‘deliver’.

For removing the stem words here we are using PORTER-STEMMER algorithm.

4.1.1.2.1 Porter-Stemmer Algorithm

The Porter stemming algorithm (or ‘Porter stemmer’) is a process for removing the commoner morphological and inflexional endings from words in English. Its main use is as part of a term normalisation process that is usually done when setting up Information Retrieval systems.

Removing suffixes by automatic means is an operation which is especially useful in the field of information retrieval. In a typical IR environment,one has a collection of documents, each described by the words in the document title and possibly by words in the document abstract. Ignoring the issue of precisely where the words originate, we can say that a document is represented by a vector of words, or \terms\. Terms with a common stem will usually have similar meanings, for example:

CONNECT
CONNECTED
CONNECTING
CONNECTION
CONNECTIONS

STEPS:

Step 1 : Gets rid of plurals and -ed or -ing suffixes

Step 2: Turns terminal y to i when there is another vowel in the stem

Step 3: Maps double suffixes to single ones: -ization, -ational, etc.

Step 4: Deals with suffixes, -full, -ness etc.

Step 5: Takes off -ant, -ence, etc.

Step 6: Removes a final -e

Examples

Step 1: Possesses --> possess

Ponies --> poni

Operatives --> operative

Step 2: Coolly --> coolli

Furry --> furri

Fry --> fry

Step 3: Rational --> rational

Optional --> option

Possibly --> possibli --> possible

Step 4: Authenticate --> authentic

Predicate --> predic

Felicity --> feliciti --> felic

Step 5: Precedent --> preced

Operational --> operate --> oper

Fable --> fable

Step 6: Parable --> parabl

Fate --> fate (cvc)

Controllable --> controll --> control

4.1.2 Collaborative Filtering

Collaborative filtering (**CF**) is a technique used by recommender systems. Collaborative filtering has two senses, a narrow one and a more general one.

In narrower sense, collaborative filtering is a method of making automatic predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating).

In the more general sense, collaborative filtering is the process of filtering for information or patterns using techniques involving collaboration among multiple agents, viewpoints, data sources, etc

Compute the term weights for each term t in the vocabulary based on the classical tf, idf weighting.

The term frequency factor represents the number of occurrences of term t in document d.

The document frequency factor dft represents the number of documents in the repository that contain term t.

N is to the total number of documents in the repository.

idf refers to the inverse document frequency, which is computed using a logarithm in order to dampen the effect of the document frequency factor in the overall term weight.

In collaborative filtering process, if previously fixed bug reports are textually similar with the current bug report, then the files that have been associated with the similar reports may also be relevant for the current report. It has been observed in that a file that has been fixed before may be responsible for similar bugs. For example, an Eclipse bug report about incorrect menu options for parts that is not closeable. The feature computes the textual similarity between the text of the current bug report and the summaries of all the bug reports there is not much historical information that can be used for computing features that are based on collaborative filtering or the file revision history. In particular, there is less opportunity for exploiting duplicated bug reports.

4.1.3 Feature Analysis

The overall set of 19 features used in the ranking model is summarized in Table

Table 6.1: Feature Set

FEATURE	SHORT DESCRIPTION	Q-dependent?
ϕ1	Surface lexical similarity	Yes
ϕ2	API-enriched lexical similarity	Yes
ϕ3	Collaborative filtering score	Yes
ϕ4	Class name similarity	Yes
ϕ5	Bug-fixing	Yes

	recency	
ϕ6	Bug-fixing frequency	Yes
ϕ7	Summary-class names similarity	Yes
ϕ8	Summary-method names similarity	Yes
ϕ9	Summary-variable names similarity	Yes
ϕ10	Summary-comments similarity	Yes
ϕ11	Description-class names similarity	Yes
ϕ12	Description-method names similarity	Yes
ϕ13	Description-variable names similarity	Yes
ϕ14	Description-comments similarity	Yes
ϕ15	In-links = # of file dependencies of s	No
ϕ16	Out-links = # of files that depend on s	No
ϕ17	Page Rank score	No
ϕ18	Authority score	No

ϕ19	Hub score	No

As shown in the last column in the table, we distinguish between two major categories of features:

Query dependent

These are features **ϕi(r,s)** that depend on both the bug report **r** and the source code file **s**. A query dependent feature represents a specific relationship between the bug report and the source file, and thus may be useful in determining directly whether the source code file s contains a bug that is relevant for the bug report r.

Query independent.

These are features that depend only on the source code file, i.e., their computation does not require knowledge of the bug report query. As such, query independent features may be used to estimate the likelihood that a source code file contains a bug, irrespective of the bug report.

4.4 Ranking

The resulting ranking function is a linear combination of features, whose weights are automatically trained on previously solved bug reports using a learning-to-rank technique. The ranking approach to the problem of mapping source files to bug reports that enables the seamless integration of a wide diversity of features exploiting before fixed bug reports as training examples for the planned ranking model in conjunction with a learning-to-rank technique.

Given a bug report r ranking is computed as follows:

1) Rank all source code files *s* based on their scores *f(r,s)* as computed by the LR system using all 19 features .

2) Within each source file, rank all its methods *m* based on their lexical similarities with the bug report *sim(r,m).*

3) Eliminate from the ranking all methods m for which $sim(r,m) < t$ i.e., their lexical similarity with the bug report is below a pre-defined threshold t

4.5 SVM

"Support Vector Machine" (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems.

Two types of classification

4.5.1 Linear classification

In this algorithm, plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, perform classification by finding the hyper-plane that differentiate the two classes very well (look at the below snapshot).

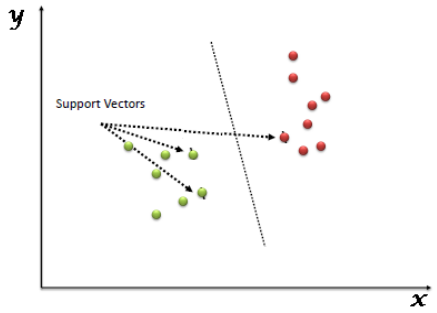


Fig. 3 Linear Classification

Support Vectors are simply the co-ordinates of individual observation. Support Vector Machine is a frontier which best segregates the two classes (hyper-plane/ line).

4.5.2 Non-Linear Classification

A way to create nonlinear classifiers by applying the kernel trick to maximum-margin hyperplanes. The resulting algorithm is formally similar, except that every dot product is replaced by a nonlinear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space. The transformation may be nonlinear and the transformed space high dimensional, although the classifier is a hyperplane in the transformed feature space, it may be nonlinear in the original input space.

Kernel methods owe their name to the use of kernel functions, which enable them to operate in a high-dimensional, *implicit* feature space without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all pairs of data in the feature space. This operation is often computationally cheaper than the explicit computation of the coordinates. This approach is called the "**kernel trick**". Kernel functions have been introduced for sequence data, graphs, text, images, as well as vectors.

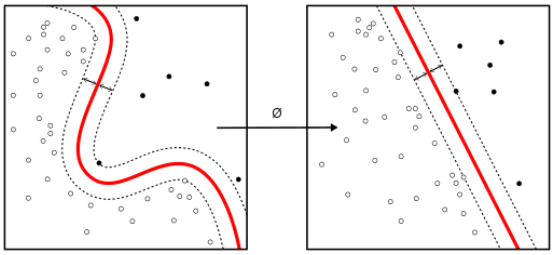
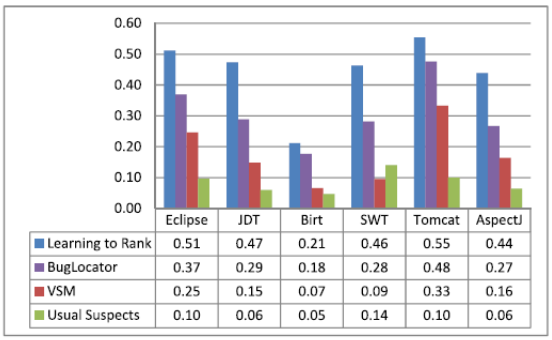


Fig. 4 Non-Linear Classification

It is noteworthy that working in a higher-dimensional feature space increases the generalization error of support vector machines, although given enough samples the algorithm still performs well.

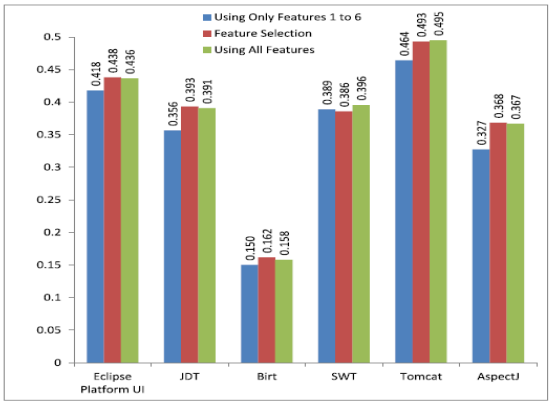
The model of the training task as a classification in which bug reports and files are assigned to multiple topics, we directly train our model for ranking, which we believe is a better match for the way the model is used.

V. RESULT & ANALYSIS

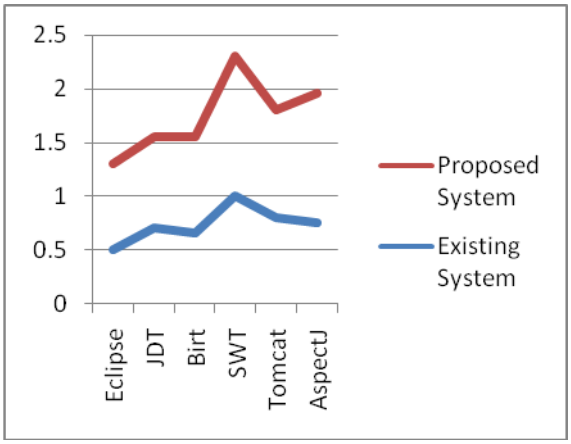


	Eclipse	JDT	Birt	SWT	Tomcat	AspectJ
Learning to Rank	0.51	0.47	0.21	0.46	0.55	0.44
BugLocator	0.37	0.29	0.18	0.28	0.48	0.27
VSM	0.25	0.15	0.07	0.09	0.33	0.16
Usual Suspects	0.10	0.06	0.05	0.14	0.10	0.06

graph.1. comparison of four methods



graph.2. comparison of features



graph.3. comparison of existing & proposed system

VI. CONCLUSION AND FUTURE WORK

To locate a bug, developers use not only the content of the bug report but also domain knowledge relevant to the software project. Here I introduced a learning-to-rank approach that emulates the bug finding process employed by developers. The ranking model characterizes useful relationships between a bug report and source code files by leveraging domain knowledge, such as API specifications, the syntactic structure of code, or issue tracking data.

The proposed adaptive ranking approach is generally applicable to software projects for which there exists a sufficient amount of project specific knowledge, such as a comprehensive API

documentation and an initial number of previously fixed bug reports. Furthermore, the ranking performance can benefit from informative bug reports and well documented code leading to a better lexical similarity, and from source code files that already have a bug-fixing history. SVM will classify the bug reports, that helps the developer to classify the bug belongs to which category and helps him/her to fix it early. In phase-1 I have implemented first two modules and it showed a good performance improvement, hope the SVM will give good accuracy.

REFERENCES

[1] Aizat Azmi, Ahmad Amsyar Azman, Sallehuddin Ibrahim, and Mohd Amri Md Yunus, "Techniques In Advancing The Capabilities Of Various Nitrate Detection Methods: A Review", International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 2, June 2017, pp. 223-261.

[2] Tsugunosuke Sakai, Haruya Tamaki, Yosuke Ota, Ryohei Egusa, Shigenori Inagaki, Fusako Kusunoki, Masanori Sugimoto, Hiroshi Mizoguchi, "Eda-Based Estimation Of Visual Attention By Observation Of Eye Blink Frequency", International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 2, June 2017, pp. 296-307.

[3] Ismail Ben Abdallah, Yassine Bouteraa, and Chokri Rekik , "Design And Development Of 3d Printed Myoelctric Robotic Exoskeleton For Hand Rehabilitation", International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 2, June 2017, pp. 341-366.

[4] S. H. Teay, C. Batunlu and A. Albarbar, "Smart Sensing System For Enhanceing The Reliability Of Power Electronic Devices Used In Wind Turbines", International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 2, June 2017, pp. 407- 424

[5] SCihan Gercek, Djilali Kourtiche, Mustapha Nadi, Isabelle Magne, Pierre Schmitt, Martine Souques and Patrice Roth, "An In Vitro Cost-Effective Test Bench For Active Cardiac Implants, Reproducing Human Exposure To Electric Fields 50/60 Hz", International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 1, March 2017, pp. 1- 17

[6] P. Visconti, P. Primiceri, R. de Fazio and A. Lay Ekuakille, "A Solar-Powered White Led-Based Uv-Vis Spectrophotometric System Managed By Pc For Air Pollution Detection In Faraway And Unfriendly Locations", International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 1, March 2017, pp. 18- 49

[7] Samarendra Nath Sur, Rabindranath Bera and Bansibadan Maji, “Feedback Equalizer For Vehicular Channel”, International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 1, March 2017, pp. 50- 68

[8] Yen-Hong A. Chen, Kai-Jan Lin and Yu-Chu M. Li, “Assessment To Effectiveness Of The New Early Streamer Emission Lightning Protection System”, International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 1, March 2017, pp. 108- 123

[9] Iman Heidarpour Shahrezaei, Morteza Kazerooni and Mohsen Fallah, “A Total Quality Assessment Solution For Synthetic Aperture Radar Nlfm Waveform Generation And Evaluation In A Complex Random Media”, International Journal on Smart Sensing and Intelligent Systems., VOL. 10, NO. 1, March 2017, pp. 174- 198

[10] P. Visconti ,R.Ferri, M.Pucciarelli and E.Venere, “Development And Characterization Of A Solar-Based Energy Harvesting And Power Management System For A Wsn Node Applied To Optimized Goods Transport And Storage”, International Journal on Smart Sensing and Intelligent Systems., VOL. 9, NO. 4, December 2016 , pp. 1637- 1667

[11] YoumeiSong,Jianbo Li, Chenglong Li, Fushu Wang, “Social Popularity Based Routing In Delay Tolerant Networks”, International Journal on Smart Sensing and Intelligent Systems., VOL. 9, NO. 4, December 2016 , pp. 1687- 1709

[12] Seifeddine Ben Warrad and OlfaBoubaker, “Full Order Unknown Inputs Observer For Multiple Time-Delay Systems”, International Journal on Smart Sensing and Intelligent Systems., VOL. 9, NO. 4, December 2016 , pp. 1750- 1775

[13] Rajesh, M., and J. M. Gnanasekar. "Path observation-based physical routing protocol for wireless ad hoc networks." International Journal of Wireless and Mobile Computing 11.3 (2016): 244-257.

[14]. Rajesh, M., and J. M. Gnanasekar. "Congestion control in heterogeneous wireless ad hoc network using FRCC." Australian Journal of Basic and Applied Sciences 9.7 (2015): 698-702.

[15]. Rajesh, M., and J. M. Gnanasekar. "GCCover Heterogeneous Wireless Ad hoc Networks." Journal of Chemical and Pharmaceutical Sciences (2015): 195-200.

[16]. Rajesh, M., and J. M. Gnanasekar. "CONGESTION CONTROL USING AODV PROTOCOL SCHEME FOR WIRELESS AD-HOC NETWORK." Advances in Computer Science and Engineering 16.1/2 (2016): 19.

[17]. Rajesh, M., and J. M. Gnanasekar. "An optimized congestion control and error management system for OCCEM." International Journal of Advanced Research in IT and Engineering 4.4 (2015): 1-10.

[18]. Rajesh, M., and J. M. Gnanasekar. "Constructing Well-Organized Wireless Sensor Networks with Low-Level Identification." World Engineering & Applied Sciences Journal 7.1 (2016).

[19] L. Jamal, M. Shamsujjoha, and H. M. Hasan Babu, "Design of optimal reversible carry look-ahead adder with optimal garbage and quantum cost," International Journal of Engineering and Technology, vol. 2, pp. 44–50, 2012.

[20] S. N. Mahammad and K. Veezhinathan, "Constructing online testable circuits using reversible logic," IEEE Transactions on Instrumentation and Measurement, vol. 59, pp. 101–109, 2010.

[21] W. N. N. Hung, X. Song, G. Yang, J. Yang, and M. A. Perkowski, "Optimal synthesis of multiple output boolean functions using a set of quantum gates by symbolic reachability analysis," IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 25, no. 9, pp. 1652–1663, 2006.

[22] F. Sharmin, M. M. A. Polash, M. Shamsujjoha, L. Jamal, and H. M. Hasan Babu, "Design of a compact reversible random access memory," in 4th IEEE International Conference on Computer Science and Information Technology, vol. 10, june 2011, pp. 103–107.

[23] Dr. AntoBennet, M, Sankar Babu G, Suresh R, Mohammed Sulaiman S, Sheriff M, Janakiraman G ,Natarajan S, "Design & Testing of Tcam Faults Using T_H Algorithm", Middle-East Journal of Scientific Research 23(08): 1921-1929, August 2015 .

[24] Dr. AntoBennet, M "Power Optimization Techniques for sequential elements using pulse triggered flipflops", International Journal of Computer & Modern Technology , Issue 01 ,Volume01 ,pp 29-40, June 2015.