Assisting bug Triage in Large Open Source Projects Using Approximate String Matching

Amir H. Moin and Günter Neumann
Language Technology (LT) Lab.
German Research Center for Artificial Intelligence (DFKI)
Saarbrücken, Saarland, Germany
{*amir.moin, neumann*}*@dfki.de*

***Abstract*—In this paper, we propose a novel approach for assisting human bug triagers in large open source software projects by semi-automating the bug assignment process. Our approach employs a simple and efficient n-gram-based algorithm for approximate string matching. We propose and implement a recommender prototype which collects the natural language textual information available in the summary and description fields of the previously resolved bug reports and classifies that information in a number of separate inverted lists with respect to the resolver of each issue. These inverted lists are considered as vocabulary-based expertise and interest models of the developers. Given a new bug report, the recommender creates all possible n-grams of the strings, evaluates their similarities to the available expertise models concerning a number of well-known string similarity measures, namely Cosine, Dice, Jaccard and Overlap coefficients. Finally, the top three developers are recommended as proper candidates for resolving this new issue. Experimental results on 5200 bug reports of the Eclipse JDT project show weighted average precision value of 90.1% and weighted average recall value of 45.5%.**

***Keywords*-software deployment and maintenance; semi-automated bug triage; approximate string retrieval; open source software;**

I. Introduction

Open source software projects often provide their developer and user communities with an open bug repository for reporting the software defects in order to be tracked by developers and users. Each bug report usually undergoes a triage process in which a group of developers, known as triagers, check whether it contains sufficient amount of information for the developers, whether it is not a duplicate of a previously reported bug and if the bug is reported at the right place. Only if the bug report passes these filters successfully then they would assign a priority degree and a severity degree to it from the business perspective and from the technical point of view, respectively. Last but not least, the triagers should assign each bug report to a developer in order to hopefully resolve the issue. This latter part, i.e. bug report assignment defines the scope of our work.

In large open source projects where hundreds or thousands of developers are collaborating with each other the main question is which person would be the best candidate for fixing a newly reported bug. Human triagers often take developers' fields of expertise and interest into consideration in order to reduce the bug resolution cost for the project. However, since the number of bug reports and the rate of their production could become very large, the bug triage process itself might become labor intensive when performed manually.

In recent years, there have been a number of valuable contributions in order to address this problem. We overview most of the works which we are aware of them in section 4 briefly. One common approach for semi-automated bug assignment is to employ a supervised machine learning algorithm through which a classifier is trained and used to categorize new bug reports. In such text categorization problems, documents are usually considered as word vectors and the term weights (Term Frequency-Inverse Document Frequencies, abbreviated as TF-IDF) are calculated. Support Vector Machines (SVMs) have turned out to be the best supervised machine learning algorithms in applying such an approach.[1]

Another common approach which has already been applied to several subfield of the bug triage problem area such as the duplicate bug report recognition problem [2] is using document indexing and information retrieval techniques. This latter approach is somehow closer to our position. However, we clearly make several distinctions between our work and those contributions. Our approach is novel in that we consider n-gram representations for strings rather than considering tokens. Furthermore, we perform approximate string matching (a.k.a. approximate vocabulary look-up) with a flexible similarity threshold parameter rather than a fixed exact match. Concerning the application domain, this brings a noticeable capability. For example, one does not have to concern about the misspelled words in the bug reports too much anymore. Last but not least, we do not directly consider the similarity between the vector representation of two text documents, but instead the approximate similarity between the n-gram representations of strings are taken into account.

Regardless of the applied techniques, almost all prior work in this field could be categorized in three main groups-based

on the source of information which they use in order to extract the developers' expertise and interest models. One major source of information is the previously resolved bug reports which are archived in the repositories of bug tracking systems like Bugzilla[1], Jira[2], etc.[3][4][1][5] Another one is the log information of the source code revision control system, i.e. comments of developers in each source code commit (for example to the Subversion[3] or CVS[4] source code repository).[6][4][5] Finally, one could explicitly use the vocabulary of the committed source code by each developer as a means of extracting such an expertise model.[7]

Our recommender is efficient and powerful enough to benefit from all of the above mentioned resources in large scales in order to make its recommendation more precise while performing fast. However, in the current implementation we have only included the first one, i.e. the previously resolved bug reports in the open bug repository.

This paper makes two main contributions:

1) It proposes a novel approach which employs a simple and efficient approximate string matching algorithm [8] in order to find appropriate developers who are more likely to have sufficient expertise and interest levels to resolve a new issue.
2) It provides an implementation of the proposed approach as well as experimental results on a large dataset of 5200 bug reports from the Eclipse Java Development Tools (JDT) project including the achieved information retrieval evaluation metrics, namely, Accuracy, weighted average precision, weighted average recall and F-measure for the top three developers recommendation.

The paper is structured as follows: Section 2 presents the proposed approach. In section 3, we implement our recommender prototype and present experimental results. Related work in this field are discussed in section 4. Finally, conclusion and possible future work are stated in section 5.

II. The Proposed Approach

The core idea is to apply an n-gram vocabulary-based approach with approximate string matching. We start with a dataset of previously resolved bug reports where each instance contains the free text available in the summary and description of the bug report and the name of the developer who has resolved the bug. For each developer p_i we automatically extract a vocabulary from all bug reports assigned to p_i. This gives us a set D of vocabularies d_i ($1 \leq i \leq n$, where n is the number of different developers p_i). Then, for a new bug report, we automatically extract its vocabulary d_j, and compute its overlap with D. We compute a ranked list of names of developers $\{p_i\}$ with respect to the degree of overlap of d_j and the corresponding vocabulary $d_i \in D$, and finally recommend the top three developers as possibly good candidates for resolving this new bug.

Our approach is semi-automated, since a human triager would need to choose one of the recommended developers by our system in order to finally assign the bug to that person.

A. Background: SimString

Since our approach is vocabulary-based, we need to address the problem of spelling variations (e.g., spelling errors like "Pyhton" instead of "Python" or name variations like "FileOpenDialog" vs. "'File Open' Dialog"). Furthermore, since we are extracting a number of different large-scale vocabularies, we need fast and scalable approximate string matching algorithms.

SimString[5] is a C++ library that uses the CPMerge algorithm [8] for fast approximate string matching. The idea is to construct an n-gram-based inverted index from the entries of a vocabulary (basically a list of strings of instances of some common semantic class). The n-gram representation of a string s is just the set of all substrings of length n of s. For example, for n=3, the n-grams for the string "python" are {pyt, yth, tho, hon}, and for the string "pathon" they are {pat, yth, tho, hon}. Matching is then realized by defining a similarity function that applies a τ-overlap join between the n-gram representation of a query and the n-gram inverted index, where τ represents the degree of similarity. For example, for the cosine function, when using τ=1, the two above entries would not match, but setting τ=0.7 they would match.

For our purpose, the most important properties of SimString are:

1) It allows finding matches in a collection of millions of entries in only a few milliseconds, e.g., using cosine similarity function and a similarity threshold of 0.7 only about 1 millisecond is needed for a query on standard PC hardware.
2) Beside the cosine similarity function, SimString utilizes additional well-known similarity functions, namely, Jaccard, Dice and Overlap coefficients.
3) When constructing the n-gram-based inverted index, the exact value of n can be parametrized.
4) SimString uses efficient disk-based hashing for maintaining the inverted index, and hence, it has a very good memory footprint.

We will now describe in detail how we are employing SimString in our recommender prototype which is called Approxricom.

[1] http://www.bugzilla.org/
[2] http://www.atlassian.com/software/jira/overview
[3] http://subversion.apache.org/
[4] http://www.nongnu.org/cvs/
[5] http://www.chokkan.org/software/simstring/

B. Our Bug Resolver Recommender: Approxricom

A Bug report (a.k.a. issue) in an open bug repository (a.k.a. open issue (bug) tracking system) often has a life-cycle. It is initially created by any registered user or developer. In this very beginning stage, its status is set to NEW. Later during its life-cycle its status might change to other possible values such as ASSIGNED, RESOLVED DUPLICATE, RESOLVED INVALID, RESOLVED WORKSFORME, RESOLVED WONTFIX, RESOLVED FIXED, VERIFIED FIXED or CLOSED FIXED. No matter the bug is currently at which state, it always resides in the open bug repository of the open source project. Our recommender only uses issues which are currently in the RESOLVED FIXED, VERIFIED FIXED and CLOSED FIXED states in order to create its vocabulary-based expertise and interest model.

Furthermore, each bug report is a structured file which consists of a number of fields in addition to its current state, such as the bug ID, summary, description, product, component, importance, assignee, reporter, date of report, date of the latest modification, CC list (list of people who are interested in getting updated about this issue), comments, etc. Among all these fields, our recommender is only concerned with the bug ID, product name, status, summary and description of the bug reports.

As depicted in figure 1, the workflow of our recommender comprises three main parts:

1) Collecting and preprocessing the required data including the bug IDs, the resolver developer for each bug and the summary and description text of each bug report in order to create the vocabulary-based expertise and interest models. (corresponds to step 1 in figure 1)
2) Creating the n-gram-based inverted lists of the non-trivial terms in summaries and descriptions of previously resolved bug reports. (corresponds to step 2 in figure 1)
3) Recommending the three most similar vocabulary databases to each query which contains a number of non-trivial terms extracted from the summary and description of a newly reported bug. (corresponds to steps 3 to 5 in figure 1)

The second and third parts are done through employing SimString and specifying the similarity measure, the similarity threshold and the size of n-grams for string representations.

Concerning the first part, our recommender connects to the open bug repository of an open source project and retrieves a large set of successfully resolved bug reports for a specific product. For example, in case of the Eclipse bug repository, there exist many products such as Platform, JDT, CDT, etc. It is clear that we would like to recommend proper bug resolvers working within one product and not in any of the products which are hosted there and might not necessarily relate to each other.

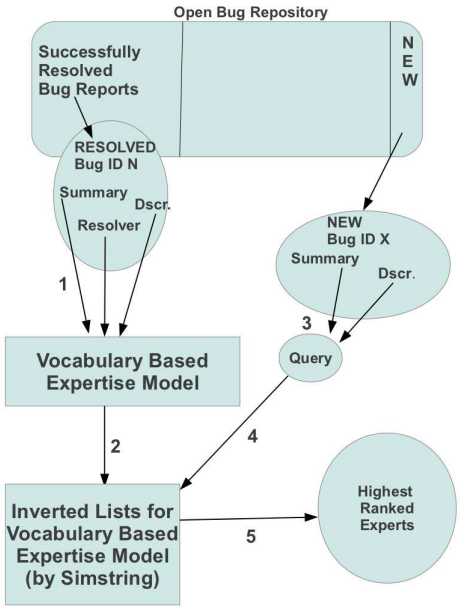

Figure 1. Bug Resolver Recommendation by Approxricom using SimString

Moreover, we perform data preparation by setting all characters to lower case and removing all stop words from the bug reports.

The second and third parts of the recommendation workflow deal with employing a fast approximate string matching algorithm to store the inverted lists (inverted indexes) in an efficient way and later retrieve them with respect to the similarity level of the query, i.e. the vocabulary of the new bug report to each of those lists which we actually consider them as the vocabulary-based expertise and interest models of developers. Given a new bug report as a query, our recommender assigns a score to each vocabulary-based expertise model database, i.e. to each developer[6] and finally recommends the top three databases (developers) as proper candidates for this query, i.e. for the resolution of this new bug report.

III. Implementation & Experimental Results

We implement our prototype using Java, C++ and Python. The interface for interacting with the open bug repository is developed in Java using XML Remote Procedure Call (XML RPC). The core of Approxricom which employs SimString is developed in C++. Finally, we take advantage of Python for our data preparation using the python-based Natural Language Toolkit (NLTK)[7].

[6]Each vocabulary-based expertise model database corresponds to one developer.

[7]http://nltk.org/

Table I
EVALUATION USING WELL-KNOWN INFORMATION RETRIEVAL METRICS (W.A. STANDS FOR WEIGHTED AVERAGE)

Fold /	Accuracy /	W.A. Precision /	W.A. Recall /	F-measure
1	50.19	85.8	46.02	59.91
2	50.86	83.16	44.39	57.89
3	46.73	75.05	42.62	54.37
4	49.8	86.35	41.46	56.03
5	51.92	94.98	44.81	60.89
6	55	96.85	50.4	66.3
7	57.11	97.16	50.25	66.24
8	57.11	93.76	53.29	67.95
9	55.57	100	42	59.15
10	53.37	88.14	39.82	54.86
Average	52.76	90.12	45.50	60.35

A. Parameters

The approximate string matching algorithm which we have employed in Approxricom, i.e. SimString is configurable with various parameters. In particular, one could set the parameter *n* for n-gram size, the similarity measure for vocabulary look-up (Cosine, Dice, Jaccard and Overlap coefficients) and the degree of flexibility in the approximate matching, known as the similarity threshold which is between 0 and 1.0 (1.0 corresponds to exact matching instead of approximate matching).

After doing some experiments with various values of configuration parameters, we decided to use the following setting for our recommender:

1) The size of n-grams (n) is 5. In other words, all possible 5-grams are created for string representations.
2) All these four similarity measures are taken into account: Cosine, Dice, Jaccard, Overlap.
3) The similarity threshold is set to 0.95. While maintaining a high degree of accuracy (since it is close to 1.0) this value prevents the recommender from trivially doing exact string matching which is in contrast to our core idea of approximate string matching in order to handle spelling errors and different writing forms.

However, we believe that further empirical research work is required to investigate the role of the above mentioned parameters in the achieved evaluation results and probably find out the best parameter settings concerning the application.

B. Scoring Strategy

In order to rank our inverted lists based on the degree of their similarity to a query, we store the maximum number of similarities which are reported by each of the four similarity functions, Cosine, Dice, Jaccard and Overlap coefficients. This maximum number is considered as the score of that inverted list given a query string and a similarity threshold.

C. Experimental Results

In order to evaluate our approach we did experiments on 5200 resolved bug reports of the Eclipse JDT project. After doing data preparation as stated in the previous section, we performed a 10 folded random cross validation on this dataset which led to the results of table I (values are in percent). As shown in the table, we have achieved average weighted precision of 90.12% and average weighted recall of 45.50%. Moreover, the accuracy is 52.76% on average.

We consider the top three high-ranked developers which are recommended by our bug resolver recommender as proper candidates for resolving a new bug report. Therefore, it is clear that if the developer who has fixed an unseen test instance bug report in real world is present among the three top-ranked developers which are recommend by Approxricom, we consider this as a success. Otherwise, it is considered as a failure.

The True Positive (TP) rate, also known as recall, measures how much part of a specific class is captured. In other words, the TP rate (recall) is the proportion of the instances which are retrieved as class A, among all instances which indeed have class A.

Precision is the proportion of the instances which indeed have class A, among all those instances which are retrieved as class A.

Since there is often a trade-off between precision and recall, it is common to measure the final performance via a mixture of both, called F-Measure.

$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall}$

Finally, Accuracy is the proportion of the total number of correctly classified instances among all instances.

According to our experimental results and the evaluation metrics, it is obvious that our proposed approach is competitive with other related approaches which are discussed in the next section.

IV. RELATED WORK

One related contribution is Mockus and Herbsleb's Expertise Browser[6]. They have introduced a tool which uses source code change data from a revision control repository to determine appropriate experts to work on various elements of software projects.

Cubranic and Murphy [3] have trained a Bayesian classifier using descriptions of fixed bug reports in open bug repositories as machine learning features, and names of developers as class labels. They have reported the accuracy value of 30% for Eclipse projects. However, they fairly believe that even this level of classification accuracy could significantly lighten the load that the human triagers face in large open source projects.

Canfora and Cerulo[4] have proposed an information retrieval approach. They have used textual descriptions of fixed change requests stored in open source software repositories, both bug repository and revision control system (Bugzilla and CVS), to index developers and source files as documents in an information retrieval system. The indices are utilized to suggest the most appropriate developers to resolve a new

change request. They have reported recall values of 10-20% and 30-50% for Mozilla projects and KDE, respectively.

Their approach is similar to ours in that they also index documents and developers to solve an information retrieval problem. However, from their very brief explanation it is clear that they do not apply any approximate n-gram-based string retrieval algorithm.

Anvik et al.[1] have presented an approach which expands on Cubranic and Murphy's previous work[3]. They have used additional textual information of bug reports beyond the bug description, to form the machine learning features. They have also applied a non-linear Support Vector Machines (SVMs) and C4.5 algorithms in addition to the Naive Bayes classifier which their predecessor work had used. They have compared the achieved results using the three classifiers and found SVM the best one for this purpose.

Anvik and Murphy[5] have presented an empirical evaluation of two approaches for determining who has the implementation expertise for a bug report using data from two types of repositories: The source repository check-in logs and the bug repository. They have found that different repositories are useful in different situations, based on what is wanted.

Jeong et al.[9] have introduced a graph model based on Markov chains to capture bug tossing[8] history. The model could be used both to reveal team structures to find suitable experts for a new task and also to better assign developers to bug reports. Their model has reduced bug tossing by up to 72% and improved the accuracy of automatic bug assignment by 23% comparing the common manual bug triage process.

Baysal et al.[10] have presented a theoretic framework for automating assignment of bug-fixing tasks with an emphasis on learning developer preferences. They have proposed to apply a vector space model to recommend experts for resolving bug reports based on the level of expertise, current workload and preference of developers which are inferred from the previously fixed bugs by each developer. Implementation of their novel model has remained as future work.

Matter et al.[7] have stood in an outstanding position, since they model developer expertise by explicitly comparing the vocabulary found in the source code contributions of developers with the vocabulary of bug reports. The advantage of their approach is that no previous activity of a developer in the current project is necessarily required. Instead, any prior activity of a developer through interacting with a source code revision control system would be enough to model his or her expertise. They report 33.6% top-1 precision and 71% top-10 recall.

Bhattacharya and Neamtiu [11] have improved triaging accuracy and reduce tossing path lengths by employing several techniques such as refined classification using additional attributes and intra-fold updates during training.

Finally, Servant and Jones [12] have presented a new technique which automatically selects the most appropriate developers for fixing the fault represented by a failing test case. Their technique is the first to assign developers to execution failures without the need for textual bug reports. On one hand, they do fault localization to map the current software bug to the corresponding line of code. On he other hand, they perform history mining to find out who has committed that buggy line of code. They have reported 81% of success (accuracy) for the top-three developer suggestions.

While their contribution is a valuable one, it is basically different than our point of view. Because, they try to find who has introduced a bug to the software so that he fixes the bug himself. In contrast, we believe that the developer who has caused a bug might not necessarily be the best one to resolve it.

V. Conclusion and Future Work

In this paper, we presented a novel approach to semi-automate the bug assignment task of human triagers in large open source projects using an expert recommender system. We create vocabulary-based expertise and interest model of developers based on the history of their contributions in resolving previous bug reports. Using a fast and efficient algorithm for approximate string matching, we store inverted lists of the mentioned vocabulary-based models and efficiently query them to find the most similar ones to a new bug report.

Our promising experimental results transparently show that our proposed approach is competitive with other approaches and our recommender system performs more efficiently than prior semi-automated bug triagers.

Our approach is novel in that it uses approximate n-gram-based string matching. Therefore, it is capable of handling spelling errors and different forms of writing. Moreover, it is extremely fast and efficient.

Like other related work, the implicit assumption is that for a resolved bug report, the person who has finally resolved the issue in practice is the one whose name is mentioned as the assignee of that bug. While this assumption is in most cases true, there exist open source projects in which this is not always the case.

Similar to many other related work, one limitation of our approach is that it needs the history of previously resolved bug reports in order to be able to perform well. Furthermore, the prototype which we have implemented currently works only with the Bugzilla open bug repositories which support the XML RPC protocol. Extending our prototype to support more bug tracking systems and evaluating our approach with other open source projects could be done as future work.

Applying this approach on other parts of the bug triage problem such as duplicate bug reports recognition remains as

[8]Bug tossing refers to the process of re-assignment of bug reports among the developers of an open source project.

a possible future work. In addition, empirical research work is needed to find the best parameter configurations for the approximate string matching algorithm as well as the effect of those parameter values on the achieved values for the information retrieval evaluation metrics. Further, we believe that the scoring strategy which we have proposed in section 3 is itself a matter of further research work. It might turn out that one could rank the databases based on the results of the similarity measures in a smarter way.

REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Morphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[2] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368151

[3] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, 2004.

[4] G. Canfora and L. Cerulo, "How software repositories can help in resolving a new change request," in *Proceedings of the workshop on Empirical Studies in Reverse Engineering*, 2005.

[5] J. Anvik and G. C. Morphy, "Determining implementation expertise from bug reports," in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, 2007.

[6] A. Mockus and J. D. Herbsleb, "Expertise browser: A quantitative approach to identifying expertise," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2002.

[7] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 131–140. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069491

[8] N. Okazaki and J. Tsujii, "Simple and efficient algorithm for approximate dictionary matching," in *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, Beijing, China, August 2010, pp. 851–859. [Online]. Available: http://www.aclweb.org/anthology/C10-1096

[9] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2009.

[10] O. Baysal, M. W. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC)*, 2009.

[11] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609736

[12] F. Servant and J. A. Jones, "Whosefault: automatic developer-to-fault assignment through fault localization," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 36–46. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337228