Developments in Duplicate Bug Reporting Process in a Software

Ahmad Saad Khan
North Carolina State University
Raleigh NC
akhan7@ncsu.edu

Neha Kale
North Carolina State University
Raleigh NC
nkale@ncsu.edu

Sekharan Natarajan
North Carolina State University
Raleigh NC
smnatara@ncsu.edu

ABSTRACT

Testers or users submit bug reports to identify various issues with systems. Bug repositories are the central storage for the bug reports. Sometimes two or more bug reports correspond to the same defect and thus gives duplicate bug reports. To address the problem with duplicate bug reports, a person called a triager needs to manually label these bug reports as duplicates, and link them to their master reports for subsequent maintenance work.

However, in practice there are considerable duplicate bug reports sent daily and requesting triagers to manually label these bugs could be highly time consuming.

We do have modern bug tracking systems that guide the maintenance activities of software developers. The utility of these systems is hampered by an excessive number of duplicate bug reports - in some projects as many as a quarter of all reports are duplicates. A system that automatically classifies duplicate bug reports as they arrive could save developer time. In this paper, we review the procession in the development of such systems during the period of 2010-2016 by examining the related literature. We examine how the various methods proposed by different papers contribute towards achieving the goal of automatic duplicate bug report detection.

Keywords

Defect Localization; Abstract Syntax Tree (AST); Within-Project Defect Prediction (WPDP); Bug repositories; Bug Reports; Information Retrieval

1. INTRODUCTION

Modern software systems are highly complex and hence software bugs are widely prevalent. As software development projects get bigger, so does the corresponding list of bugs which creep up in these project. If a bug has been found in one area of the source code, then it is more likely that other bugs will also be found in areas near the existing bug. This is known as bug localization. To manage and keep track of bugs and their associated fixes, bug tracking systems like Bugzilla has been proposed and are widely adopted. Defect reporting is an integral part of a software development, testing and maintenance process. Typically, bugs are reported to an issue tracking system which is analyzed by a Triager (who has the knowledge of the system, project and developers) for performing activities like: quality check to ensure if the report contains all the useful and required information, duplicate detection, routing it to the appropriate expert for correction and editing various project-specific metadata and properties associated with the report (such as current status, assigned developer, severity level and expected time to closure). It has been observed that often a bug report submitted by a tester is a duplicate (two bug reports are said to be duplicates if they describe the same issue or problem and thereby have the same solution to fix the issue) of an existing bug report.

Typically, a bug report contains textual description of the issue and the method of reproducing the bug. It is a structured record consisting of many fields. Commonly, they include summary, description, project, submitter, priority and so forth. The figure below shows a typical bug report. Each field carries a different type of information. For example, summary is a concise description of the defect problem while description is the detailed outline of what went wrong and how it happened. Both of them are in natural language format. Other fields such as project, priority try to characterize the defect from other perspectives. A status field in a bug is a representation of the different stages in a lifecycle of a bug.

Studies show that the percentage of duplicate bug reports can be up-to 25-30% [17] [13] [9]. To address this issue there have been a number of studies that try to partially automate the triaging process. There are two general approaches: one is by filtering duplicate reports preventing them from reaching the triagers [18], the other is by providing a list of top-related bug reports for each new bug report under investigation [19] [2]. Developer time and effort are consumed by the triage work required to evaluate bug reports, and the time spent fixing bugs has been reported as a useful software quality metric. Modern software engineering for large projects includes bug report triage and duplicate identification as a major component. Duplicate bug reports could potentially provide different perspectives to the same defect potentially enabling developers to better fix the defect in a faster amount of time. Still there is a need to detect bug

reports that are duplicate of one another.

2. MOTIVATION

The process of bug reporting is still not mature and is uncoordinated and ad-hoc. It is very common that the same bugs could be reported more than once by different users. Hence, there is often a need for manual inspection to detect whether the bug has been reported before. If the incoming bug report is not reported before then the bug should be assigned to a developer. But if other users have reported the bug before then the bug should be classified as a "duplicate" and should be attached to the original first - reported "master" bug report. There are two challenges related to bug data that may affect the effective use of bug repositories in software development tasks and they are "large scale" and "low quality". On one hand, due to the daily-reported bugs, a large number of new bugs are stored in bug repositories. Taking an open source project, Eclipse, as an example, an average of 30 new bugs were reported to bug repositories per day in 2007 [10]; from 2001 to 2010, 333,371 bugs were reported to Eclipse by over 34,917 developers and users [12] [20].

The Mozilla programmers reported in 2005 that "every day, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [11]. This is seeming to be the most time consuming step of handling a bug and alleviating the burden of the triagers has gained a lot of importance over the past decade, in the field of automated software engineering. A lot of effort has been put in deploying an automated triaging system in the industry

Thus recognizing duplicate reports is an important problem that, if solved, would enable developers to fix bugs faster, and prevent them from wasting time by addressing the same bug multiple times. Classifying bug reports as duplicates by applying some of the Data Mining techniques of "textual similarity" and "information retrieval" has been a very popular approach.

Among several automated detection approaches, text-based information retrieval (IR) approaches have been shown to outperform others in term of both accuracy and time efficiency. However, those IR-based approaches do not detect well the duplicate reports on the same technical issues written in different descriptive terms.

Previous approaches that are used to find duplicate bugs were based on similarity score of vector space representation. There is still much room for improvement in terms of accuracy of duplicate detection process. In this paper discriminative models for information retrieval are built to detect duplicate bug reports more accurately which the authors claim would improve duplicate bug detection accuracy by 43%.

3. DATASETS

Most of the studies have used the bug repositories of open source projects - OpenOffice, Mozilla and Eclipse as their data for conducting their study. These repositories are very popular because these projects are all open source and have a large repository of bug reports and are freely accessible. These projects are also very different from one another in terms of purposes, users and implementation languages, and thus help in generalizing the conclusions of the experiments.

In [3] Android bug reports submitted from November 2007 to September 2012 were used as the dataset. After filtering out unusable bug reports the authors were left with 37,236 reports out of which 1063 were duplicates.

In [14], Amoui et al. applied the Duplicate Defect Detection Framework which they build to the Blackberry bug repository in addition to the OpenOffice, Mozilla and Eclipse repositories.

4. RELATED WORK

There has been lot of relate work on duplicate bug detection in the past. One of the pioneering studies on duplicate bug report detection is by Runeson et al. [16]. Their approach first cleaned the textual bug reports via natural language processing techniques âĂŞ tokenization, stemming and stop word removal. The remaining words were then modeled as a vector space, where each axis corresponded to a unique word. Each report was represented by a vector in the space. The value of each element in the vector was computed by the formula on the tf value of the corresponding word. After these vectors were formed, they used three measures âĂŞ cosine, dice and jaccard âĂŞ to calculate the distance between two vectors as the similarity of the two corresponding reports. Given a bug report under investigation, their system would return top-k similar bug reports based on the similarities between the new report and the existing reports in the repository.

A case study was performed on defect reports at Sony Ericsson Mobile Communications, which showed that the tool was able to identify 40% of duplicate bug reports. It also showed that cosine outperformed the other two similarity measures. Mining software repositories is an emerging field that has received significant research interest in recent times.

Hiew et al. presents an approach based on grouping similar reports in the repository and deriving a centroid of the clustered reports [**?**]. An incoming report (represented as a document vector) is then compared to the centroids of bug report groups to compute a similarity score for detecting duplicates based on a predefined threshold value. One of the unique features of the approach presented by Hiew et al. is pre-processing of bug reports to create a model (model updated as new bug reports arrived) consisting bug-report groups and their respective centroid which is then used for similarity computations.

5. BASELINE RESULTS

Sometimes two or more bug reports corresponded to the same defect. To address the problem with duplicate bug reports, a person called a triager needed to manually label these bug reports as duplicates, and link them to their "master" reports for subsequent maintenance work. However, in practice there are considerable duplicate bug reports sent daily; requesting triagers to manually label these bugs could be highly time consuming.

Figures 1 and 2 display bug report 50900 and the corresponding fixed/buggy file InteropService.java (for brevity,

only comments are shown in Figure 2). The report is about a software defect in which incoming synchronization tasks were not triggered properly in a server. We found that the developers fixed the bug at a single file InteropService.java by adding code into two methods processIncoming and processIncomingOneState to handle a stale data by refetching. As shown, both bug report 50900 and the buggy file InteropService.java describe the same problematic functionality of the system: the "synchronization" of "incoming data" in the interop service. This faulty technical aspect (was described and) could be recognized via the relevant terms, such as sync, synchronization, incoming, interop, state, schedule, etc. Considering the bug report and the source file as textual documents, we could consider this technical aspect as one of their topics. This example suggests that the bug report and the corresponding buggy source files share the common buggy technical topic(s). Thus, detecting the common technical topic(s) between a bug report and a source file could help in bug file localization.

Bug Report #50900
| Summary: Error saving state returned from update of external object; incoming sync will not be triggered.
Description: This showed up in the server log. It's not clear which interop component this belongs to so I just picked one of them. Seems like the code run in this scheduled task should be able to properly handle a stale data by refetching/retrying. |

Figure 1: Bug report

InteropService.java |
|// Implementation of the Interop service interface. // Fetch the latest state of the proxy |
|// Fetch the latest state of the sync rule |
|// Only return data from last synchronized state|
|// If there is a project area associated with the sync rule,|
|// Get an advisable operation for the incoming sync|
|// Schedule sync of an item with the state of an external object|
|// The result of incoming synchronization (of one item state).|
|// Use sync rule to convert an item state to new external state.|
|// Get any process area associated with a linked target item....|
|// For permissions checking, get any process area associated with the target item. If none, get the process area of the sync rule.|
|// return an instance of the process server service...|
|// Do incoming synchronization of one state of an item.|
|public IExternalProxy processIncoming (IExternalProxyHandle ...) {.....} |

Figure 2: Fixed Java file

6. IMPLEMENTATION

We started with paper [2] which had very high citations. This paper enlightened us about few of the methods used for duplicate bug detection. We further went on to read 3 papers backwards and 4 papers forward, with respect to this paper, that is papers from 2010 - 2011 and 2013 - 2016, with 2 papers from 2010. These papers gave us further understanding of the problem and also made us understand various methods and techniques used.

7. EVOLUTION

Until 2010, several techniques have been proposed using various similarity based metrics to detect candidate duplicate bug reports for manual verification. Automating triaging has been proved challenging as two reports of the same bug could be written in various ways. There is still much room for improvement in terms of accuracy of duplicate detection process. In [8], Sun et al. leveraged recent advances on using discriminative models for information retrieval to detect duplicate bug reports more accurately. They have validated their approach on three large software bug repositories from Firefox, Eclipse, and OpenOffice and showed that their technique could result in 17âĂŞ31%, 22âĂŞ26%, and 35âĂŞ43% relative improvement over state-of-the-art techniques in OpenOffice, Firefox, and Eclipse datasets respectively using commonly available natural language information only.

In [8], they propose a discriminative model based approach that further improves accuracy in retrieving duplicate bug reports by up to 43% on real bug report datasets. Different from the previous approaches that rank similar bug reports based on similarity score of vector space representation, they developed a discriminative model to retrieve similar bug reports from a bug repository. They use a classifier to retrieve similar documents from a collection. Then they build a model that contrasts duplicate bug reports from non-duplicate bug reports and utilize this model to extract similar bug reports, given a query bug report under consideration. Menzies and Marcus also suggested a classification based approach to predicting the severity of bug reports.

Their basic workflow for training the discriminative model is shown in Figure 3

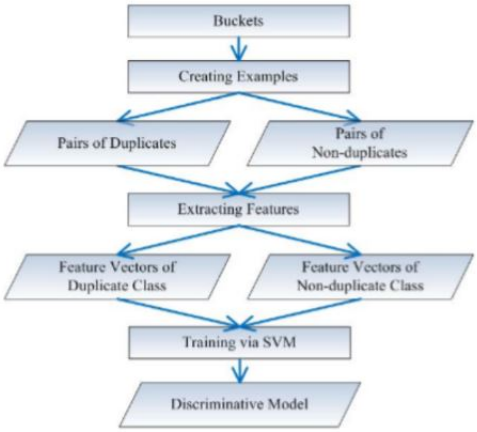

Figure 3: Workflow

In [4], Nguyen et al. propose BugScout, an automated approach to help developers reduce such efforts by narrowing the search space of buggy files when they are assigned to address a bug report. BugScout assumes that the textual contents of a bug report and that of its corresponding source code share some technical aspects of the system which can be used for locating buggy source files given a new bug report. Instead of text based discriminative model, they developed a specialized topic model that represents technical aspects as topics in the textual contents of bug reports and source files, and correlates bug reports and corresponding buggy files via their shared topics. BugScout can recommend buggy files correctly up to 45% of the cases with a recommended ranked list of 10 files. BugScout also correlates the topics in both source files and bug reports, and uses topics as a random variable in its model which made it outperformed other approaches based on LDA+VSM.

In [5], Nguyen et al. introduced DBTM, a duplicate bug report detection approach that takes advantage of both IR-based features and topic-based features. This is different from the model discriminative model used by Sun et al. in [8]. DBTM models a bug report as a textual document describing certain technical issue(s), and models duplicate bug reports as the ones about the same technical issue(s). Trained with historical data including identified duplicate reports, it is able to learn the sets of different terms describing the same technical issues and to detect other not-yet-identified duplicate ones. Evaluation on real-world systems shows that DBTM improves the state-of-the-art approaches by up to 20% in accuracy. To support the detection of duplicate bug reports, they specifically developed a novel topic model, called T-Model.

They also used BM25F, an advanced document similarity function based on weighted word vectors of documents.

Figure 4 shows the accuracy with varied number of topics:

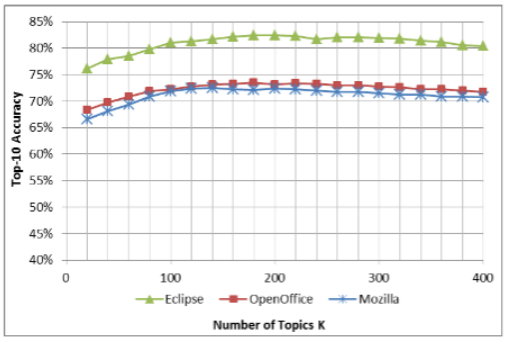

Figure 4: Accuracy with varied number of topics

The accuracy comparison of various other algorithms and DBTM is shown in Figure 5, 6 and 7.

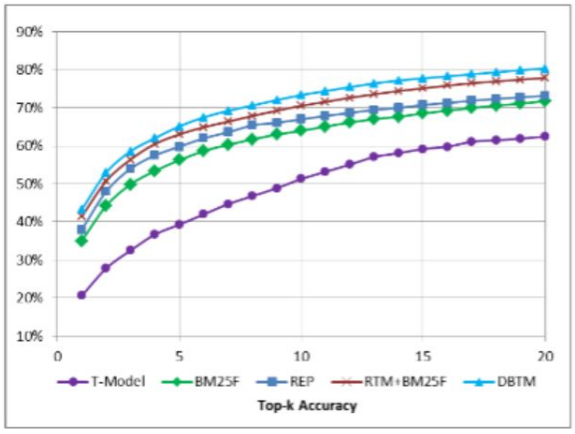

Figure 5: Accuracy comparison in Mozilla

In [7], Sun et al. proposed a retrieval function (REP) to measure the similarity between two bug reports. It fully utilizes the information available in a bug report including

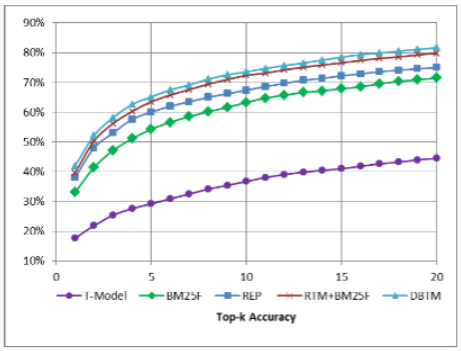

Figure 6: Accuracy comparison in Open-Office

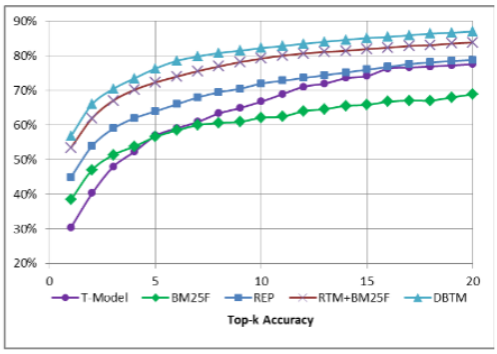

Figure 7: Accuracy comparison in Eclipse

not only the similarity of textual content in summary and description fields, but also similarity of non-textual fields such as product, component, version, etc. For more accurate measurement of textual similarity, they also extended BM25F used in [5] - an effective similarity formula in information retrieval community, specifically for duplicate report retrieval. Lastly they also use a two-round stochastic gradient descent to automatically optimize REP for specific bug repositories in a supervised learning manner.

Their approach is built upon BM25F model. BM25F is meant for facilitating the retrieval of documents relevant to a short query. In duplicate bug report detection, given a bug report (which is a rather lengthy query), all related bug reports need to retrieved. They also extend BM25F model to better fit duplicate bug report detection problem by extending the original model to better fit longer queries. Aside from the performance improvement, REP also significantly reduces the runtime compared to our past approach that uses SVM. Fig 2 shows the time needed to finish an experiment run for each dataset.

For retrieval performance of REP, compared to the previous works based on SVM, it increases recall rate by 10-27%, and MAP by 17-23%; REP performs with recall rate of 37-71% ($1 < k < 20$), and MAP of 46% which is considerably better than the results mentioned in [8].

In [14], Amoui et al came up with a different approach to

TABLE VII
OVERHEAD OF *SVM* AND *REP* (IN SECONDS)

	OpenOffice	Mozilla	Eclipse	Large Eclipse
REP	139	1603	277	7037
SVM	3408	19411	4267	> 2 days

Figure 8: Overhead of SVM and REP

look into the problem of duplicate bug detection. So far most of the study have been primarily on improving search and classification algorithms but the model themselves were not capable of learning. In this paper, the authors took into consideration the domain and characteristics of a software project for effective and high quality duplicate defect detection. In view of this, they needed a feedback system to assess the search quality and tune parameters for each software product and hence developed the Duplicate Defect Detection (DDD) framework. A high-level view of the framework is shown in figure 9.

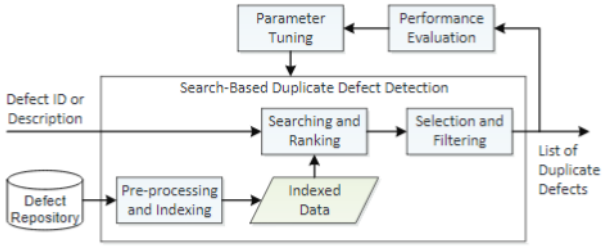

Figure 9: High-level framework view

DDD is composed of a customized search-engine and a feedback loop for evaluating and tuning the search quality for each project and its user requirements. In this section, we first describe the search process and then we elaborate on the parameter tuning and evaluation process of this framework.

The experimental studies allow us to analyze the impact of each tuning parameter in duplicate-defect detection performance for the selected datasets. In this study, we identify a set of parameters that are usually left out in search-based software engineering approaches, while their default values are not necessarily optimal for the input dataset. We also show that some parameters can greatly impact the search quality, but their optimal value may not be the same for all datasets and their impact is not constant. For example, among all tuning parameters, a particularly notable parameter was the Maximum Document Frequency. ItâĂŹs optimal value for BlackBerry dataset is 5% while for Mozilla Firefox is 20%.

In [15], Corley et al proposed a range of metrics based on the topic distribution of the issue reports, relying only on data taken directly from bug reports as shown in figure 10. In particular, they introduce a novel metric that measures the shared topic between two topic-document distributions. This paper details the evaluation of this group of pair-based metrics with a range of machine learning classifiers. They also demonstrate that the proposed metrics show a significant improvement over previous work, and conclude that the simple metrics they proposed should be considered in future studies on bug report de-duplication, as well as for more general natural language processing applications.

The results of this paper demonstrate the descriptive power of the suite of attributes they had proposed. They have also shown an increase in accuracy, AUC, and Kappa statistics over the metrics of Alipour et al [1] when applied to the Android repository. However, their study simply extends the analysis by Alipour et al. There is good reason to believe that the metrics used by Alipour et al., when combined with ours, could further improve duplicate bug detection. The effectiveness and simplicity of the metrics proposed in this study make them good candidates for futures studies on duplicate bug reports. The practices of splitting the summary and description attributes when computing similarity and of using the shared topic distance measure (which is a novel distance measure) are particularly recommended. Finally, the use of Bagging to aid in classification should be considered, as it provided a small increase in accuracy. A future study which utilized these metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further test their effectiveness. Their results suggest that this suite of metrics could significantly improve state-of-the-art top-k approaches to bug de-duplication

Table 1: Attributes for Pairs of Bug Reports

lenWordDiffSum *lenWordDiffDes*	Difference in the number of words in the summaries or descriptions
simSumControl *simDesControl*	Number of shared words in the summaries or descriptions after stemming and stop-word removal, controlled by their lengths
sameTopicSum *sameTopicDes* *sameTopicTot*	First shared identical topic between the sorted distribution given by LDA to each summary, description, or combined summary and description
topicSimSum *topicSimDes* *topicSimTot*	Hellinger distance between the topic distributions given by LDA to each summary, description, or combined summary and description
priorityDiff	{same-priority, not-same}
timeDiff	Difference in minutes between the times the bugs were submitted
sameComponent	Four-category attribute: {both-null, one-null, no-null-same, no-null-not-same}
sameType	{same-type, not-same}
class	{dup, not-dup}

Figure 10: New terms

In [3], Aggarwal et al. propose a method to partially automate the extraction of contextual word lists from software-engineering literature. Evaluating this software-literature

context method on real-world bug reports produces useful results that indicate this semi-automated method has the potential to substantially decrease the manual effort used in contextual bug de-duplication while suffering only a minor loss in accuracy.

Labelled LDA is very time-intensive, whereas the software-literature context method results in huge time savings for the developers by many orders of magnitude. The software-literature context method for bug de-duplication is at par with unsupervised LDA while reducing the time and improving generalizability.

Anahita et al. first introduced the concept of using contextual words to improve the accuracy of bug de-duplication, in [1]. But they used Labelled LDA approach to extract contextual features. This procedure took 60 person-hours to create topic words. However, in the paper [3], the authors propose "software literature context method", where the contextual words are extracted from a generic software-literature. This approach shows a significant improvement in time with a slight decrease in accuracy of prediction. The extraction of contextual words in the current approach took only half a person-hour. The general software-engineering features extracted from a textbook published in 2001, performed only marginally worse, showing the robustness of the software-literature context method.

Anahita et al. improved upon the work of Sun et al. [8] by adding contextual features using both labelled and unlabelled LDA generated word lists to the method used by Sun et al.. They reformulated the task as detecting duplicate bug reports and showed that the use of LDA produced strong improvements in accuracy, increasing by 16% over the results obtained by Sun et al.[18]. The results obtained by Aggarwal et al. in [3] are similar to the results reported by [1] but method requires far less time and effort and is more general and easy to share. The approach used by Aggarwal et al. is different from the approach described by Anahita et al in [1] as the former approach uses word lists extracted from software literature instead of using bug reports from the software projects to extract the word lists. These software-literature context word lists reflect the software development processes, and the evolution of project, and hence depict the bug report descriptions. The word lists used as contextual information were extracted from the following sources:

1. Pressman's "Software Engineering: A Practitioner's Approach"
2. "The Busy Code's Guide to Android Development" by Murphy
3. Eclipse platform documentation
4. OpenOffice Documentation
5. Mozilla Documentation
6. Random English words English random words were used to determine that the specialized word lists were actually having a significant effect

The lists used were same as the ones used in [3]. Using these contextual word lists, the procedure proposed by Anahita et al. was followed on four bug datasets from open-source projects, Android, OpenOffice, Mozilla, and Eclipse (containing approximately 37000, 42000, 72000, and 29000 bug reports correspondingly). Similar to the previous paper, several well-known machine-learning algorithms such as Logistic Regression, Naive Bayes, Decision Trees, K-Nearest Neighbors and Rule based classifiers are applied using Weka to retrieve the duplicate bug reports and 10-fold cross validation was employed to validate the retrieval. Thus, in this study, Aggarwal et al., conclude that the software literature context method, by comparison, is much faster than the approach [1] and is marginally less accurate and has lesser kappa scores when compared to labelled LDA.

Finally in 2016 , In [6], Chen et al. use the knowledge graphs to integrate concepts and their mutual relations, using four different classification algorithms, proposed a topic Bug - report clustering algorithm. They find that on average, richness of conceptions and their correlations has significant impact on the results but the selection of machine learning algorithms has a small effect on the results. They follow an incremental approach: First they use a classification algorithm to obtain preliminary knowledge, and then get richer knowledge by clustering method. The main purpose is to verify the richness of knowledge to the importance of improving the quality of a service.

Figure 11 depicts the knowledge graph data structure

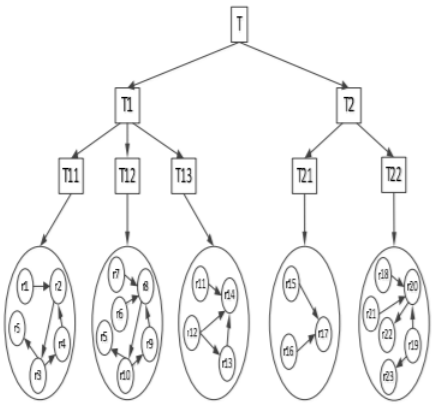

Figure 11: Knowledge graph

They use many advanced algorithms for learning like the following:

- Naive Bayes is a probabilistic technique that uses Bayes' rule of conditional probability to determine the probability that an instance belongs to a certain class. Bayes' rule states that "the probability of a class conditioned on an observation is proportional to the prior probability of the class times the probability of the observation conditioned on the class" and can be denoted as figure 12:
- Bayesian network is a graphical model which can describe the probability relation of a set of variables.

$$p(C_k|\mathbf{x}) = \frac{p(C_k)\, p(\mathbf{x}|C_k)}{p(\mathbf{x})}$$

Figure 12: Naive Bayes observations

Bayesian network is often used to express uncertainty knowledge in expert system and deal with incomplete data sets. Bayesian network can express causality and make combining domain knowledge and data. A Bayesian Network is a probabilistic model that is used to represent a set of random variables and their conditional dependencies by using a directed acyclic graph (DAG). Each node in the DAG denotes a variable, and each edge corresponds to a potential direct dependence relationship between a pair of variables. Each node is associated with a conditional probability table (CPT) which gives the probability that the corresponding variable takes on a particular value given the values of its parents.

- C4.5 is a very famous decision tree generation algorithm, which is mainly based on ID3 algorithm improvement. The decision tree generated by the C4.5 algorithms can be used for classification, for this reason, C4.5 is also known as statistical classifier. The C4.5 algorithm builds a decision tree based on the attributes of the instances in the training set. A prediction is made by following the appropriate path through the decision tree based on the attribute values of the new instance. C4.5 builds the tree recursively in a greedy fashion.

- Support Vector Machine (SVM) is one of the main machine learning techniques for high dimensional sparse data sets. For text categorization, SVM can get higher accuracy. SVM need to constantly increase the training vectors, so a SVM is essentially a quadratic programming problem. The SVM [10] is a supervised classification algorithm that finds a decision surface that maximally separates the classes of interest. That is, the closest points to the surface on each side are as far as possible from the decision surface. Four kinds of kernel functions are commonly used: Linear, Polynomial, Gaussian Radial Basis Function (RBF) and Sigmoid.

Figure 13 depicts the results of their study

8. NEGATIVE PATTERNS

In [8], the authors say that with big increase of recall rates, the runtime overhead of their detection algorithm is also higher than past approaches. In the largest dataset Firefox in the experiment run of their approach, the initial cost to detect a duplicate report is one second. However, as the training set continually grows and the repository gets increasingly large, the detection cost also increases over time.

The major overhead is due to the fact that we consider 54 different similarity features between reports which is way ahead of the number of features used by previous models.

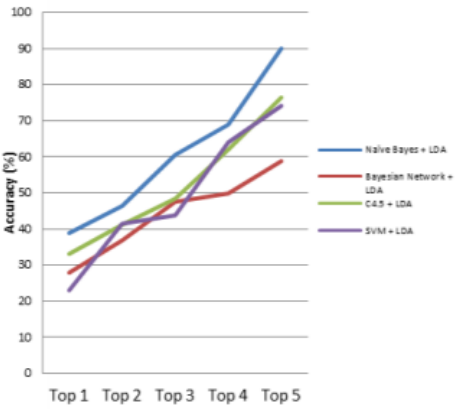

Figure 13: Results from the knowledge graph

Consequently, SVM will need more time to build a discriminative model.

9. THREATS TO VALIDITY

In [8], experiments were only performed on 4 systems. They used the same library as used in previous tools for our reimplementation which the authors themselves said is not an efficient implementation.

In [15] Because more pairs are represented by large buckets of duplicates, there is a selection bias towards bug reports in larger groups rather than smaller ones.

Paper [3], says that "By observing the properties of defect repositories, we could raise the hypothesis that for some parameters there exists a global optimal parameter that is independent from its input dataset". However, the authors also say that they cannot support this hypothesis at that time and list the parameters for which there is a fixed optimal value regardless of the input dataset and user requirements.

10. OUR COMMENTARY

In this section we give our comments on what we think about some of the papers mentioned in the previous sections.

The method mentioned in paper [8] considers all new incoming bugs as duplicates of master report. The reason behind this assumption is not explained clearly. We think that if a new bug with more features come in to the bug repository then it should be considered as a primary bug or at least added as bug reports to the master report. Master report should always have the most relevant and highest amount of information about a particular bug because this can make future comparisons easier and prevent losing import new bug that come into the system.

In paper [5], The authors wrote about an interesting case study where DBTM detected the same bug occurring in very different usage scenarios. It would have been great if they had explained it in more detail about how well DBTM is able to perform in more such scenarios. We are interested in this case because a lot of applications which use share libraries

tend to produce bugs with different root causes even though they have been cause by a module within the shared library.

11. CONCLUSION

In this paper we have covered the evolution research in the field of Mining Software Repositories and Data Mining in Software Engineering, particularly with respect to the topic - Detection of Duplicate Bug Reports. We found that with each passing year, experts of the field came up with new approaches and techniques to improve the accuracy of duplicate bug detection. Getting high accuracy in this domain is a challenge as all the bugs are reported in natural language and even though the bugs are duplicate of each other, they could be reported easily. In the first paper published in 2010, the authors just used normal text based comparison algorithms and a little bit of machine learning algorithms for duplicate bug detection. Then the focus shifted to using Topic models and Natural language processing techniques to classify and detect duplicate bugs and the last paper in 2016 is focusing on using artificial intelligence for detecting duplicate bugs by making the system context aware and adding feedback loops.

In our opinion, the duplicate bug detection system has evolved tremendously. Being able to shortlist a few from thousands of bugs automatically which then the triager can mark as duplicates is incredible. Having said that, the system still has a vast scope of improvement. The process of duplicate bug detection is not completely automated as of now. There is still lots of manual intervention required. The bug reporting system itself is manual and contains a lot of text in natural language. System generated reports of a crash or a standardization of bug reporting techniques can improve the consistency in bug reporting. We see that system generated reports have become very common these days. Combination of such reports and the techniques discussed throughout the paper can yield a higher accuracy in detection of duplicate reports.

12. OUR RECOMMENDATION

In [14], the authors have employed one-dimensional optimization technique for finding optimal parameter values can be extended to more advanced heuristics (e.g., Genetic Algorithms, Differential evolution and so on) as we think that some of the parameters and not mutually exclusive and hence optimizing these parameters individually might negate the improvements made on other parameters. They can also explore the search space more efficiently, and suggest pareto optimal values when optimizing for conflicting objectives (i.e., measures).

In [8], as a future work, the authors can investigate the utility of paraphrases in discriminative models for potential improvement in accuracy. They have developed a technique to extract technical paraphrases and are currently investigating their utility in improving detection of duplicate bug reports. An interesting direction is to incorporate response threads to bug reports as further sources of information. The other interesting direction is adopting pattern-based classification to extract richer feature set that enables better discrimination and detection of duplicate bug reports.

13. REFERENCES

[1] E. S. A. Alipour, A. Hindle. A contextual approach towards more accurate duplicate bug report detection. *Proceedings of Working Conference on Mining Software Repositories*, 2013.

[2] C. S. A. Nyugen, D. Lo. Duplicate bug report detection with a combination of information retrieval and topic modeling. *International Conference on Automated Software Engineering*, 2012.

[3] K. Aggarwal. Detecting duplicate bug reports with software engineering domain knowledge. *Journal of Software: Evolution and Process*, 2016.

[4] J. A.-K.-H. V. N. T. N. N. Anh Tuan Nguyen, Tung Thanh Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. *IEEE Transactions on Software Engineering*, 2011.

[5] T. N. N.-D. L. C. S. Anh Tuan Nguyen, Tung Thanh Nguyen. Duplicate bug report detection with a combination of information retrieval and topic modeling. *IEEE International Conference on Automated Software Engineering*, (27).

[6] S. J. Chen Jingliang, Ming Zhe. A data-driven approach based on lda for identifying duplicate bug report. *IEEE 8th International Conference on Intelligent Systems (IS)*, 2016.

[7] S.-C. K. J. J. Chengnian Sun, David Lo. Towards more accurate retrieval of duplicate bug reports. *IEEE Transactions on Software Engineering*, 2011.

[8] X. W. J. J. S.-C. K. Chengnian Sun, David Lo. A discriminative model approach for accurate duplicate bug report retrieval. *ACM/IEEE International Conference on Software Engineering*, 1:45 – 54, 2010.

[9] L. Hiew. Assisted detection of duplicate bug reports. *MS Computer Science Thesis, Department of Computer Science, British Columbia, Canada*, 2003.

[10] G. C. M. J. Anvik. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology*, 20(3), 2011.

[11] G. C. M. J. Anvik, L. Hiew. Coping with an open bug repository. *eclipse âĂŹ05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology exchange*, pages 35 – 39, 2005.

[12] Z. R. W. Z. J. Xuan, H. Jiang. Developer prioritization in bug repositories. *International Conference on Software Engineering*, (34):25 – 35, 2012.

[13] G. C. M. John Anvik, Lyndon Hiew. Who should fix this bug? *International Conference on Software Engineering*, pages 361 – 370, 2006.

[14] A. A.-D. L. T. Mehdi Amoui, Nilam Kaushik. Search-based duplicate defect detection: An industrial experience. *IEEE Transactions on Software Engineering*, 4Duplicate Bug Report Detection with a Combination0(99):173 – 183, 2013.

[15] N. K. Nathan Klein, Christopher Corley. New features for duplicate bug detection. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[16] O. N. P. Runeson, M. Alexandersson. Automatic summarization of bug reports. *International Conference on Software Engineering*, 2007.

[17] G. M. Sarah Rastkar, G Murphy. Automatic

summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(99):366 – 380, 2014.

[18] e. a. Sun, Chengnian. A discriminative model approach for accurate duplicate bug report retrieval. *IEEE International Conference on Software Engineering*, 1(32), 2010.

[19] P. J. Sureka, Ashish. Detecting duplicate bug report using character n-gram-based features. *Software Engineering Conference (APSEC)*, 2010.

[20] H. Y.-R. Z. Z. W. L. Z. W. X. Xuan J, Jiang H. Towards effective bug triage with software data reduction techniques. *IEEE Transactions on Knowledge and Data Engineering*, (27):264 – 280, 2015.