Duplicate Bug Detection

Abhishek Kumar
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
akumar21@ncsu.edu

Akash Pandey
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
apandey5@ncsu.edu

***Abstract* - In a large software system, there are many users that interact with the system and report issues. Thus there are times when a bug gets reported by multiple users, resulting in duplicate bug reports. Detecting such duplicate bug reports helps reduce the triaging effort and saves developers' time in resolving the same issues. There have been many studies conducted to automate the detection of duplicate reports.**

In this study, we try to summarize the work done in this area. We will discuss the various techniques used for automated triaging as well as challenges involved with it. We will also discuss the work that has been put in to improve the accuracy of duplicate bug detection.

Keywords
Triaging, Duplicate Bug Detection, Bug tracking, Information Retrieval, Topic Modelling, Deduplication, contextual information, BM25F

INTRODUCTION

In any large software system, it is expected to have a large number of bugs. With the increase in the complexities of the system we see an increase in the defects. Finding and fixing these defects can be a challenging task. This task become more challenging with a software system that frequently evolves.

Fixing defects is one of the most important part of software maintenance. It is also one of the most costly operations in a software project. Bug tracking tools are used in almost all the large software projects. With the help of bug tracking tools like 'BugZilla' users can easily find any issues in the system and report them. This makes it easier for developers to keep track of the reported issues and to plan on bug fixing activities.

There could be many reasons why bugs are introduced in a system. Some of them could be bad design of the software architecture, mistake while coding, insufficient knowledge of the domain and incomplete or ambiguous requirement specifications. Sometimes it could be because of a side effect of a change in the requirement. It is a common practice to label bugs based on their level of criticality.

There are multiple players or stakeholders in a project who can report bugs in the tracking system. As they usually test the system independently of each other and report bugs there can be a number of bugs which are reported more than once by different or even same person. So, it can be said that bug reporting is an uncoordinated distributed process. Because of this there are instances of same defect being reported by many different stakeholders. Thus in a large software system, there is a high probability of occurrences of duplicate bugs. For example, in one of the papers authors reported that in the repository for Mozilla, it is normal to find more than 300 bugs every day that are duplicates and need manual effort to label them as such.

When a bug is reported, it is assigned to a developer for inspection and possible correction activity. The problem with having duplicate bug reports is that the same issue gets assigned to more than one developer and is clearly a waste of resources. Moreover, it also affects the maintenance cost and time for a software system. Detection of duplicate bug reports can also provide additional information about the original issue, like its current status, for eg. 'Has it been fixed yet?' Thus there is a need for a detection of duplicate bugs.

Since detection of duplicate bugs is a critical operation, there are resources that are allocated in a project for this task. Hence there is a need for inspection, manual or automated to check whether the bug has been reported more than once. If there is only one instance of the reported bug then it should be assigned to a developer for a fix. However, if there has been multiple occurrences of the same bug then this should be marked as a duplicate and in ideal case should be attached to the 'master' bug. This process is known as triaging.

As we can imagine in a large software system this will be a very time consuming task. Thus there is a need for automating this process and the papers that we have studied have suggested many approaches towards achieving an accurate triaging. What we found is that early papers on this subject focused more on the Information retrieval (IR) [13] to this problem.

As we found in our study that many researchers have approached the problem of bug deduplication using the information-retrieval tools such as BM25F. This approach generally entailed a process in which a bug report is modeled as a vector of textual features (Vector Space Model) which is computed via Term Frequency (Tf) and Inverse document frequency (Idf).

Above approach using Information Retrieval is a good approach and later research were performed to achieve a higher accuracy by building on top of the IR. Some of the papers that we read suggested applying Topic Modelling in association with textual based Information Retrieval.

MOTIVATION

With increase in the complexities of the large system, the number of defects will increase. As the business requirements in today's world is getting more and more complex, there is a need to pay special attention to the bug fixing aspect of software management. For many critical systems finding bugs early and fixing them is of paramount importance. Software maintenance generally tends to be the most costly operation in the whole software project. Thus there is a need to make sure that resources are properly utilized and not wasted.

Thus it is a common practice to use bug tracking tools to manage the issues reported during the testing and other phases in the software. These bug tracking tools are a great way to have a common interface between different stakeholders to report anything they feel is out of place. It also allows them to test the system independent of each other and to report the found defects. With this convenience of bug tracking tool, there is also the problem of duplicate bug reporting. As people interacting with the bug tracking tool often lack motivation, skills, knowledge or time to check if a bug is already reported in the system.

Thus there are duplicate bugs introduced in the bug tracking system. The problem with duplicate bugs is that if it is not carefully assessed and identified as a duplicated, it could result in a lot of wasted resources. If there are multiple instances of the same bug in the bug tracking system and let's say that these are allocated to different developers then we are wasting all the developer's time and other resources that could have been utilized elsewhere. Also, this is clearly bad for the cost of the maintenance activity of the project hence the overall cost of the software project.

There has been a lot of study done for the bug deduplication process. The ones that seems most intuitive is to use the textual comparison of the bug report description. This process is known as the textual Information Retrieval and is still widely used in large projects. For eg. BM25F is one such information-retrieval tool. This approach although good is not very accurate. Thus a lot of study has been done to improve the accuracy of such information retrieval methods. Some of those suggest approaches include taking the contextual information in association with the information retrieved, using a discriminative models for information retrieval, performing topic modelling together with information retrieval.

These studies have shown that it is an open area of research and a lot of improvement can be made in improving the accuracy of the automated deduplication of bugs in the bug tracking system. This will result in saving a lot of resource that otherwise would get wasted in either manual triaging. Also, the amount of developer's time wasted on fixing the duplicate bugs will also be saved. This will result in saving a lot of cost associated with the correction activity in the maintenance phase thus reducing the overall cost of the software project.

We also discovered that finding of duplicate bugs has other advantages other than saving cost and resources. It can also tell us more about the original bug and its current status. For eg, whether it is still open or not. Also, it can tell more about the system in general. With our study of these papers we are aiming to put together all the recent work done in this field and to summarize the different approaches that researchers have taken to solve the problem of deduplication and to guess the next steps that this research could go next.

HYPOTHESIS

In all the papers that we studied, the authors have a common hypothesis that improving the accuracy of duplicate bug detection and automated deduplication will result in the improved cost of the correction activities in the maintenance phase and thus the overall cost of the software project. There have been a lot of different experiments done using different approaches on large code repositories which are expected to have a large number of duplicate bugs. All these approach are trying to improve the accuracy of the detection of duplicate bug. Also, in some of the papers we read, it is suggested that finding these duplicate bugs will have other advantages other than reducing the cost of maintenance cost. It is also hypothesized that these detection will also help understand other details about the bugs, for eg. It's priority, its current status etc.

RELATED WORK

Traditionally IR methods like Vector Space Model (VSM) have been used for duplicate bug report detection. To improve the performance of IR models, they have been augmented with tools from NLP and adding extra features like failure traces attached to bug reports have been used. Another set of solutions to this problem have used machine learning techniques like binary classification and Support Vector Machines (SVM) have been used. Machine learning (ML) techniques take more time and are less efficient on their own. Modern IR based approaches using a techniques called BM25F outperform ML based approaches. This paper combines topic based ML and BM25F IR techniques to produce even better results.

One of the pioneers of the study of duplicate bug detection is done by Runeson et al. Their approach involved first processing the textual report of bug by using Natural Language Processing techniques such as – tokenization, stemming and removing the stop words in the report. Stop words are those words that doesn't add any value in the comparison of one report with other. For eg, - the, a, an etc. The remaining useful words were modelled as a vector space, whereas each axis corresponded to a unique word. Each bug report was thus a vector in this space. The magnitude of each element in this vector was calculated using the formula: weight(word) = 1+ log2(tf(word)). Using this they found a Cosine similarity between the reports and if the calculated similarity is above a certain threshold then the two reports are considered duplicates.

Among those who extended the work of Runeson was Wang et al. They extended his work in two dimensions. First they not only considered TF, but also IDF (Inverse document freqency). So, in their work the value of each element in a a vector corresponded to the following formula, weight (word) = tf(word) * idf (word). The other change that did was to consider execution information to detect duplicates. A case study which was based on Cosine similarity found that their approach could detect 67%-93% of duplicate bug reports by utilizing the execution trace along with natural language processing.

Jalbert and Weimer extended the work by Runeson et al. They also made a couple of extension in the approach suggested by Runeson. First they proposed a new term-weighting scheme for

the vector representation, which was weight (word) = 3 + 2* log2 (tf (word)). Secondly, they cosine similarity was modified to extract the top-k similar reports. Other than these they also made use of clustering and classification techniques to filter duplicates.

There are researches done which focus generally on bug reprts. It is suggested that duplicate reports complement to one another to help in bug fixing. Battenberg et al. [9] analyzed information to determine useful properties in a bug report. He found the mismatch between what the developer needs and what users supply in the bug report. Other researchers studied bug reports on their types, quality and severity or priority.

In the first paper [1] that we read, the author agrees that among all the automated detection approaches for detecting duplicate bugs, text-based information retrieval (IR) approaches have been outperforming others in terms of accuracy and also time efficiency. But these IR-based approaches do not detect well the duplicate reports on the same technical issues written in different descriptive terms. The solution that they introduced is called DBTM, a duplicate bug report detection. A approach that takes advantage of both Information retrieval and topic-based features. DBTM models a bug report as a textual document describing certain technical issues, and models duplicate bug reports as the ones about the same technical issue. The author used historical data to train his model including the identified duplicate reports, it is able to learn the sets of different terms describing the same technical issues and to detect other unidentified duplicate reports.

In DBTM, a bug report is considered as a textual document describing one or more technical issues/topic in a system. A duplicate bug is the one which describes the same technical issue but in different terms. The authors extended the Latent Dirichlet Allocation (LDA) [10] to represent the topic structure for a bug report. This enabled them to report the duplication relations among them. So, two duplicate bugs must describe about the shared technical issue/topic in addition to their own topics on different phenomena. The topic selection of a bug report is affected not only by the topic distribution of the project, but also by the buggy topic for which the report is intended.

The authors have run their experiments on three large code repositories. They are Eclipse, OpenOffice and Mozilla. The following figures show that they have managed to achieve an improvement in accuracy of up to 20%.

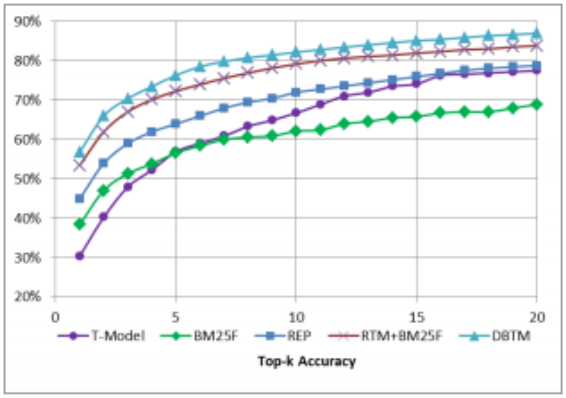


Figure 1: Accuracy Comparison in Eclipse

The figure above, no. 1 displays the accuracy comparison of DBTM against REP on Eclipse Dataset. As shown, DBTM achieves very high accuracy in detecting bug reports. For a new bug report, in 57% of the detection cases, it can correctly find the duplication by using only a single recommended bug report (i.e. master report). Within a list of top-5 resulting bug reports, it correctly detects the duplication in 76% of the cases.

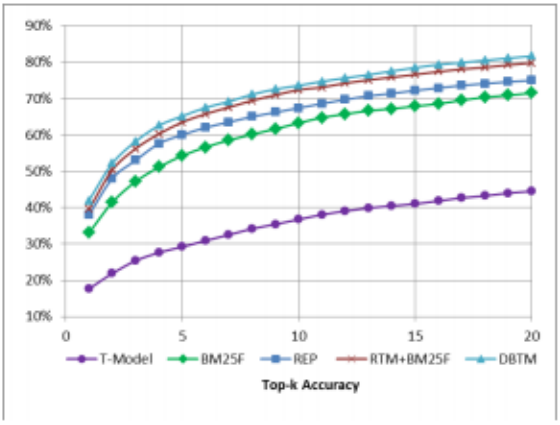


Figure 2: Accuracy Comparison in OpenOffice

These results are consistent in other experiments: OpenOffice and Mozilla. Figure 2 and 3 display the accuracy of the results on these datasets. So, DBTM consistently improves over REP with higher accuracy from 4% - 6.5% for OpenOffice and 5% - 7% for Mozilla.

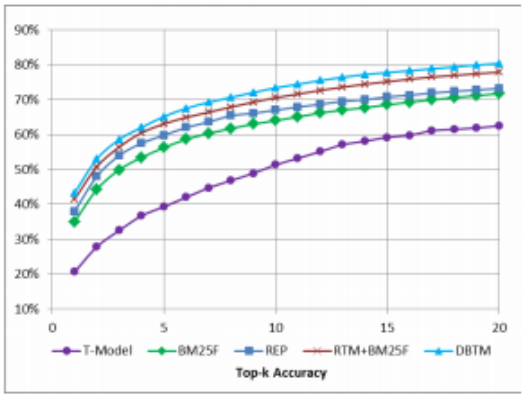


Figure 3: Accuracy Comparison in Mozilla

The second paper that we read was Relational Topic Models for Document Networks [2]. The motivation behind this paper was that the ability to predict the content of a document from its links and links from the documents has widespread applications in fields like social networks, scientific papers citations and links among web pages. Most approaches before this paper treat the content and the connections between documents separately. This paper uses a combined approach which gives it the unique ability to predict content when connection is given and vice versa. Also the proposed model can make accurate predictions for hitherto unseen data. eg. Friends for a new profile on a social network.

Relational Topic Model(RTM), is a model of documents and the links between them. The RTM models the links between each pair of documents as a binary random variable that is conditioned on their contents. This model can be used to summarize a network of documents, predict links between them, and predict words within them. They have also derived efficient inference and learning algorithms based on variational methods and evaluate the predictive performance of the RTM for large network of documents.

In RTM a document is modelled using LDA (Latent Dirichlet allocation) topic modelling technique. LDA is a generative probabilistic model popular for topic modelling to describe a corpus of documents. In its generative process, each document is allocated a Dirichlet distributed vector of topic proportions, and each word of the document is assumed drawn by first drawing a topic assignment from the overall topic distribution for document and then drawing the word from the corresponding topic distribution pertaining to that topic.

In the context of RTM a document networks is a collection of textual documents that are related to each other in some way, so that they form links between each other that can be modelled as a network. e.g. Scientific papers and their citations, web pages and hyperlinks. A variational inference is a technique used in Bayesian inference to estimate intractable quantities, usually probabilities and expectations. The authors use it estimate posterior inference, exact value of which is intractable. A family of distribution functions for latent variables are fit by varying the parameters.

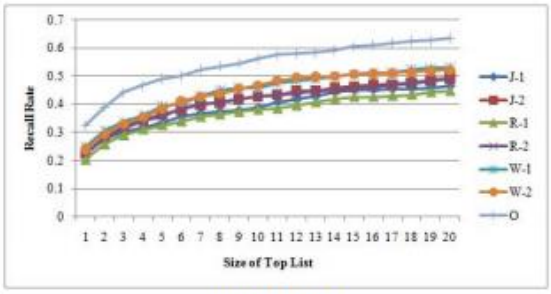
The third paper we studied was: A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval [3].

The motivation behind this paper was that the author realized that automated triaging has been proved challenging in various ways. There was still much room for improvement in terms of accuracy of duplicate detection process. So, they leveraged the recent advances on using discriminative models for information retrieval to detect duplicate bug reports on three large software bug repositories from Firefox, Eclipse, and OpenOffice.

What was different about this paper from all previous approaches that rank similar bug reports based on similarity score of vector space representation is that they developed a discriminative model to retrieve similar bug reports from a bug repository. They made use of recent advances in the information retrieval community that uses a classifier to retrieve community that uses a classifier to retrieve similar documents from a collection of documents. They built a model that contrasts duplicate bug reports from non-duplicate bug reports and utilize this model to extract similar bug reports, given a query of bug report under consideration.

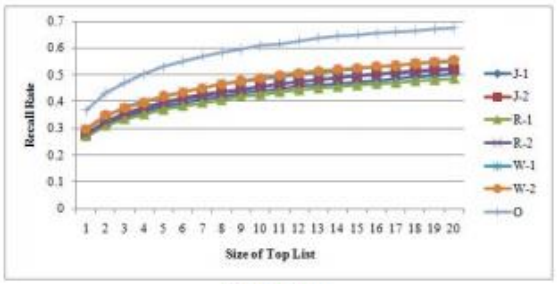
In their paper the authors discuss about introducing many more relevant features to capture the similarity between bug reports. The most impressive thing about this is that with the relative importance of each feature in this approach will be automatically determined by the model through assignment of an optimum weight. This means, that as the bug repository evolves, the discriminative model also evolves to guarantee that all weights remain optimum at all time. Thus this approach is very adaptive, robust and automated.

The authors evaluated their discriminative model approach on three large bug report datasets from large code repositories including Firefox, an open source web browser, Eclipse, a popular open source integrated development environment, and OpenOffice, a well-known open source rich text editor. The following figures show the result they get by running their model on the above mentioned projects bug report datasets compared against the state-of-the-art techniques using commonly available natural language information alone.



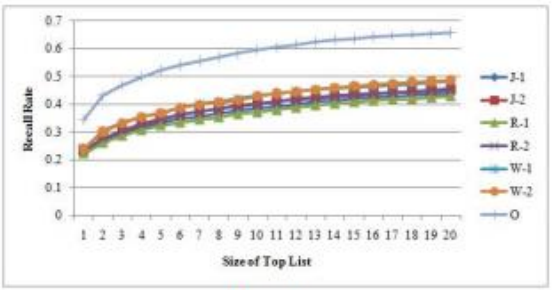
(a) OpenOffice

Figure 4: An increase in accuracy of 17-31%



(b) FireFox

Figure 5: An increase in accuracy of 22-26%



(c) Eclipse

Figure 6: An increase in accuracy of 35-43%

The author also explained about the major overhead with their approach which is due to the fact that they consider 54 different similarity features between reports. This is a major contrast from previous works which considered similarity only between summary and summary and description and description. However, this overhead does not mean that their approach is not feasible in the real world projects. The authors explain as they have experimented their approach on three real world large bug database, and since for an active software project, it is enough to consider a one year time-frame as bug reports are usually received for new software releases.

The fourth paper that we read was 'Towards More Accurate Retrieval of Duplicate Bug Reports' [4]. In this paper the author talks about how in order to identify the duplicates in the bug report more accurately, they proposed a new retrieval function (REP) to measure the similarity between two bug reports. They not only utilized the similarity of textual content in summary and description fields but also similarity of non-textual fields such as product, component, version, etc. They extended the BM25F for a more accurate measurement of textual similarity. They also used a two-round stochastic gradient descent to automatically optimize REP for specific bug repositories in a supervised learning manner.

The primary motivation of the authors behind this paper was that in order to tackle the problem of duplicate bugs reports in the bug datasets, there are a couple of options. First option is to prevent the duplicate bug report to reach the triager and the other option is to provide the triager with a list of top-k related reports for each new report. In their work the authors have shown an alternative approach with the goal of improving the accuracy of existing techniques. With a more accurate technique, triagers could be presented with better candidate duplicate bug report lists which would make his/her job easier.

The authors evaluated their approach on several large bug report datasets from large open source projects including Mozilla, Eclipse and OpenOffice and found that their technique could result in 10-27% relative improvement in recall rate. The following table lists the fields of interest for the authors in the OpenOffice bug reports. Field summary and description are in natural language text, and they referred to them as textual features, whereas the other fields try to characterize the report from other perspectives and they refer to them as non-textual features or categorical features.

TABLE I
FIELDS OF INTEREST IN AN OPENOFFICE BUG REPORT

Field	Description
Summ	*Summary*: concise description of the issue
Desc	*Description*: detailed outline of the issue, such as what is the issue and how it happens
Prod	*Product*: which product the issue is about
Comp	*Component*: which component the issue is about
Vers	*Version*: the version of the product the issue is about
Prio	*Priority*: the priority of the report, *i.e., P1, P2, P3, ⋯*
Type	*Type*: the type of the report, *i.e., defect, task, feature*

The authors have experimented with the extended BM25F and only involved textual similarity on summary and description in their experiment. The figure below shows the curve of the recall rate@k of the two similarity measures on the four datasets. It was observed that on each dataset for each different size of the top-k result list, BM25Fext performs constantly better than BM25F and it gains 4%–11%, 7%–13%, 3%–6% and 3%–5% relative improvement over BM25F on OpenOffice, Mozilla, Eclipse and Large Eclipse datasets respectively.

TABLE V
MAP OF $BM25F_{ext}$ AND $BM25F$

	OpenOffice	Mozilla	Eclipse	Large Eclipse
$BM25F_{ext}$	45.21%	46.22%	53.21%	44.22%
$BM25F$	41.92%	41.76%	51.06%	42.25%
Relative Impro.	7.86%	10.68%	4.20%	4.66%

The fifth paper that we studied was 'A Contextual Approach towards More Accurate Duplicate Bug Report Detection' [5]. The motivation behind this paper, according to the authors is that many researchers have approached the bug-deduplication problem by using the off-the-shelf information-retrieval tools, such as BM25F, so they wanted to investigate how contextual information, relying on the prior knowledge of the software quality, software architecture, and system-development (LDA) topics, can be exploited to improve bug-deduplication.

With their work the authors are trying to solve the problem of detecting duplicate bug reports automatically. They noticed that when the number of daily reported bugs for a popular software is taken into consideration, manually triaging takes a considerable amount of time and the results are unlikely to be complete. They gave an example of Eclipse, where two person-hours are daily being spent on bug triaging. There has been studies that address this issue by automating bug-report deduplication. To that end, many bug-report similarity measurements have been proposed so far which are primarily based on the textual features of the bug reports, and they also utilize natural-language processing (NLP) techniques to do textual comparison. There were also some studies which focused on the categorical features extracted from the fields of bug reports, (i.e. component, version, priority, etc).

The authors took a different approach towards improving the accuracy of detecting duplicate bug reports of a software system. Their approach exploited the domain knowledge, about the software-engineering process in general and the system specifically, to improve bug-report deduplication. Their approach is based on a hypothesis that bug reports are likely to refer to software qualities, i.e., non-functional requirements, or software functionalities. The authors utilized a few software dictionaries and word lists, exploited by prior research to extract the context implicitly in each bug report. They created a new feature by comparing the contextual word lists with the bug report and recorded the comparison results. Then they used this extracted feature along with primitive features such as description, component, type, priority etc to compare bug reports and detect duplicates.

The authors evaluated their approach on a large bug-report data-set from the Android project, which is a Linux-based operating system with several sub-projects2. They examined about 37236 Android bug reports. They utilized five different contextual word lists to study the effect of various software engineering contexts on the accuracy of duplicate bug-report detection. These five word lists are: Android architectural words [11], software Non-Functional Requirements words [12], Android topic words extracted by applying Latent Dirichlet Allocation (LDA) method [9], Android topic words extracted applying Labeled-LDA method, and random English dictionary words. To retrieve the duplicate bug reports, several well-known machine-learning algorithms are applied. To validate the retrieval approach they employed 10-fold cross validation.

Figure 7 shows the ROC curves for results of applying K-NN algorithm on the various features (all-features). The no context shows the performance of K-NN algorithm using the data generated by Sun et al's measurements (i.e. only textual and categorical measurements) which shows poor performance in comparison to the other curves. The best performance is seen for the Labeled-LDA which outperforms all the other curves. There are separate curves for the various features in this graph. And is clear from it that the addition of extra features with or without Sun et al's features improves bug-deduplication performance.

Figure 8 represents the ROC curves comparison between all the features when the author's applied C4.5 algorithm. It also indicates the performance of C4.5 on the "textual categorical" table. It clearly highlights the difference in performance of C4.5 using different contextual data-sets and its performance without using any context.

Duplicate bug reports have similar context.
Context provides different information than textual similarity.

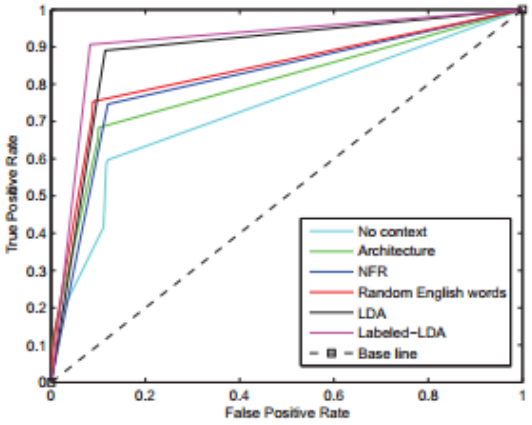


Figure 7: ROC Curve for K-NN algorithm

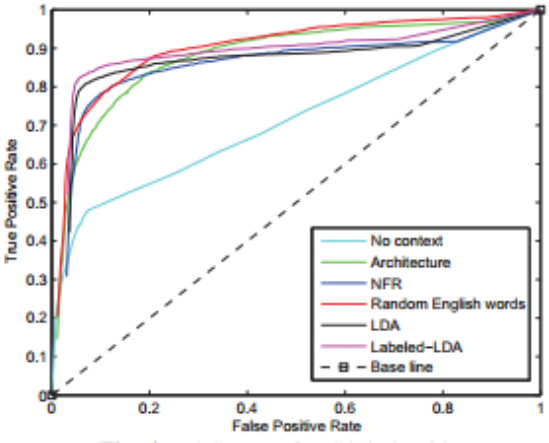


Figure 8: ROC Curve for C4.5 algorithm

The sixth paper that we studied was 'DRONE: Predicting Priority of Reported Bugs by Multi-Factor Analysis' [6]. This paper is different from all the other papers that we read because it tackles a related but different problem in the process of triaging. This paper focuses on predicting the priority of a reported bug. The motivation behind the paper is described by the authors as, among the other duties of a bug triager is also predicting the priority level to the reported bug. As resources are limited, bug reports would be investigated based on their priority levels. This is a manual effort. This paper proposes an automated approach based on machine learning that would recommend a priority level based on information available in bug reports. There are multiple factors to consider here like temporal, textual, author, related-report, severity, and product that potentially affect the priority level of a bug report. To train their discriminative model authors have extracted these factors as features and fed it to a new classification algorithm that handles ordinal class labels and imbalanced data.

The authors in their approach have leveraged the information available in the bug reports. Some of the information extracted from the bug reports are short and long descriptions of the issues users encounter while using the software system, the products that are affected by the bugs, the estimated severity of the bugs, and many more. Their approach predicts the priority level of bug reports by considering several factors that could affect the priority of the bug report. These include temporal factors, for example the bug reports that are reported at the same time, the textual content of the bug report (textual), the author of the bug report (author), related bug reports (related-report), the estimated severity of the bug(severity), and the product which the reported bug affects (product).

The authors have proposed a new machine learning approach, in particular a new classification algorithm, to create a model from the features that could predict if a bug report should be assigned a priority level P1, P2, P3, P4, or P5. They took a training set of reports along with the priority levels. From this training set, they then extract the feature values from each data point. The machine learning algorithm will then decide the likely priority level of a bug report whose priority level is to be predicted based on these feature values.

The authors named their new framework as DRONE (PreDicting PRiority via Multi-Faceted FactOr ANalysEs). The purpose of DRONE is to assist bug triagers in assigning priorty labels to bug reports. The authors also named their new classification engine as GRAY (ThresholdinG and Linear Regression to ClAssifY Imbalanced Data) which enhances linear regression with our thresholding approach to handle imbalanced bug report data. Using Linear regression the authors have modelled the relationship between the values of the various features and the priority levels of a bug reports. Linear regression considers the priority levels as ordinal values (i.e., P1 is closer to P2 than it is to P5) rather than categorical values (i.e., P1 is as different to P2 as it is to P5).

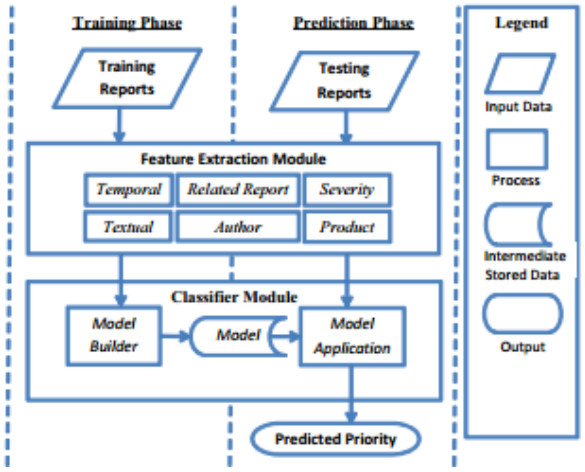


Figure 9: Framework of DRONE

Figure 9 shows the DRONE framework. As we can see it runs in two phases: training and prediction. The two main modules are feature extraction module and classification module. In the training phase, this framework takes an input a set of bug reports with known priority labels. The various features are extracted by the extraction module and these capture temporal, textual, author, related-report, severity, and product factors that potentially affect the priority level of a bug report. These features are passed to the classification module which would produce a discriminative model which helps classify a bug report with unknown priority level.

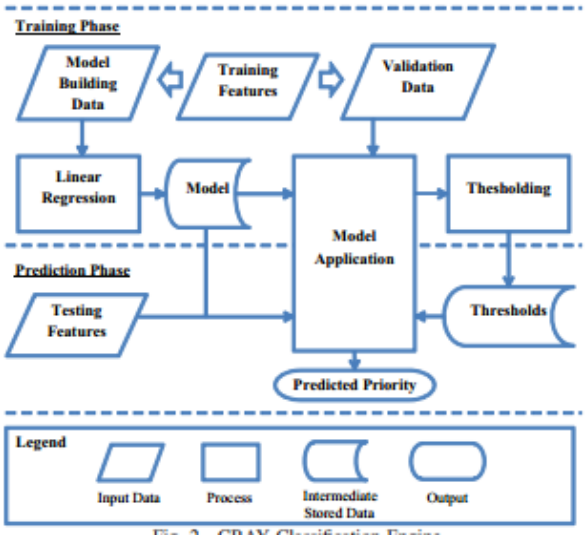


Figure 10: GRAY Classification Engine

Figure 10 shows the classification engine developed by the authors, and is named GRAY. It has two main parts: linear regression and thresholding. As the data is usually imbalanced i.e., most of the bug reports are assigned priority level P3, the authors employed a thresholding approach to calibrate a set of thresholds to decide the class labels. The linear regression is used to model the relationship between the features and the priority levels.

PRECISION, RECALL, AND F-MEASURE FOR DRONE

Priority	Precision	Recall	F-Measure
P1	41.15%	42.39%	41.76%
P2	10.92%	12.46%	11.64%
P3	91.36%	82.77%	86.85%
P4	0.24%	1.77%	0.43%
P5	4.97%	20.72%	8.01%
Average	29.73%	32.02%	29.74%

The table above shows the comparison of accuracy of DRONE vs Accuracy of Baselines. It can be noted that we can predict the P1, P2, P3, P4, and P5 priority levels by an F measure of 41.76%, 11.64%, 86.85%, 0.43%, and 8.01% respectively. The F-measures are better for P1, P2, and P3 priority levels but are worse for P4, and P5 priority levels. The authors believe in report prioritization high accuracy for high priority bugs is much more important than high accuracy for low priority bugs.

PRECISION, RECALL, AND F MEASURE FOR SEVERIS[Prio]

Priority	Precision	Recall	F-Measure
P1	0.00%	0.00%	0.00%
P2	0.00%	0.00%	0.00%
P3	88.25%	100.00%	93.76%
P4	0.00%	0.00%	0.00%
P5	0.00%	0.00%	0.00%
Average	17.65%	20.00%	18.75%

The seventh paper that we read was 'Automated Library Recommendation' [7]. The motivation for this paper is described by the authors as, there are many third party libraries available to be downloaded that can reduce the development time and make the developed software more reliable. However, developers are often unaware of suitable libraries to be used for their requirements in a project and they miss out on the benefits of using libraries. Thus, to help developers better take advantage of the available libraries, the authors propose a new technique that automatically recommends libraries to developers. Their technique takes as input the set of libraries that an application currently uses, and recommends other libraries that are most likely to be relevant.

The author's work is different from all the previous work in this field with respect to the terms of the level of granularity considered. While previous approaches make the assumption that the libraries are already known to the user, he only needs recommendations on which methods to use from the library. The author's approach does not make this assumption, and target the problem of recommending an entire library. (e.g., an entire jar file in the case of a Java project). So, in a way this suggested approach can be deployed first to recommend particular libraries that will interest developers. These results could then be fed to all the previous approaches to recommend particular methods to be used in different contexts.

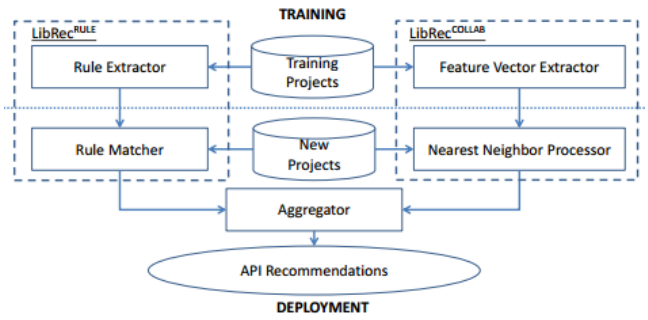


Figure 11: LibRec framework

Figure 11 shows the overall framework of our recommendation system referred to as LibRec. It has two major components: LibRecRULE and LibRecCOLLAB. LibRecRULE recommends libraries by mining library usage patterns expressed as association rules. LibRecCOLLAB makes use of the nearest-neighbor-based collaborative filtering approach to recommend libraries by investigating the set of libraries that are used by similar projects. Their framework consists of two phases: training and deployment.

To evaluate the results, authors have used recall rate@5 and recall rate@10, which are often used as evaluation measures. In their experiment they achieve recall rate@5 and recall rate@10 of 0.852 and 0.894, respectively.

The eight paper that we read was 'Automatic Recommendation of API Methods from Feature Requests' [8]. The motivation for this paper is described by the author as, developers often recieve many feature requests, to implement these features, developers can leverage various methods from third party libraries. In order to help developer by recommending the useful API method, the authors propose an automated approach that takes as input a textual description of a feature request and then recommends methods in library APIs that developers can use to implement the feature. The recommendation approach suggested learns from the records of other changes made to software systems, and then compares the textual description of the requested feature with the textual description of various API methods.

There have been previous studies which recommend code units given a requirements. These techniques, however, are not sufficient to automatically process feature requests, which typically describe high-level requirements, written in natural language. So, in their work, the authors have proposed a complementary approach that recommends relevant library methods directly from feature requests.

The suggested approach learns from a training dataset of changes made to a software system recorded in repositories (i.e., issue management systems, and version control systems). The changes in the dataset are three-fold: the textual description describing the change (text), the code before the change (pre-change), and the code after the change (post-change). The functionality of the method suggested by the author is to take as input a new textual description (text) and then recommends methods from a set of libraries to be used in the post-change code.

The ranking methodology upon the relevant methods used by the authors is two-pronged. The first method is to search for similar closed or resolved feature requests in the training data. A closed or resolved feature requests is the one that has been addressed by developers and where appropriate changes have been made to the software system. It then compares the API methods used in such requests with each other thus finding the relevance of an API method in a feature request. The second step involves measuring relevance by looking into the similarity between the textual description of the feature request and the descriptions of the API methods. Then the authors employed approach is to recover a list of potentially relevant library methods that are then recommended to the developers.

The authors evaluated their solution on feature requests stored in the JIRA issue management systems of 5 JAVA applications:

Axis2/Java, CXF, Hadoop Common, HBase, and Struts 2. The accuracy of the proposed approach is measured using the recall-rate@5 and recall-rate@10. Their experiments show that they can achieve a recall-rate@5 and recall-rate@10 of 0.690 and 0.779 respectively.

Project	Recall-Rate@5	Recall-Rate@10
Axis2/Java	0.805	0.908
CXF	0.628	0.725
Hadoop Common	0.571	0.709
HBase	0.789	0.839
Struts 2	0.657	0.713
Overall	**0.690**	**0.779**

BASELINE RESULTS

These papers related to automated duplicate bug report detection, we observed that they choose the Information Retrieval (IR) based approaches as their baseline which makes use of Natural Language Processing (NLP) technique. The pattern that we observed is that only textual information retrieved from the bug reports is a good baseline over which many different improvements and extensions have been applied to improve the accuracy of duplicate bug detection.

For the two papers that we studied that were about the recommendation of libraries and API method to the developers, the base line results were taken as rate@k to measure the effectiveness of our approach. This is a well-known measure that is used in many past studies.

CONCLUSION

In today's world, large software project will always have a lot of complexity and a lot of players involved in its different phases or lifecycle. Thus, there will always be defects introduced in the system and since there are independent stakeholders using and testing the system at different phases, there will always be duplicate bugs in the system. We have explained how having duplicate bugs in the bug database can be really bad resources utilization vise. Though, there are certain useful information that can be extracted from these duplicate bug reports that can help better understand the cause of the defect and the nature of the system.

Thus, finding duplicate bug reports is extremely useful in the maintenance of a software system. Though manual triaging is still the most accurate way to discover the duplicate issues, there has been a lot of progress in automating this detection and deduplication process. The primary way to automatically detect duplicates is through the Natural Language Processing (NLP) technique. A lot of tools have been developed that use NLP upon the bug report description to find duplicates. With the advent of machine learning techniques, and we saw many new approaches that have come forward that improves the accuracy of the automated bug detection process.

Then we also how the progress in the automated extraction of information from documents, and the various different ways to extract similarity between two documents leads to other uses in the software industry. We went through the example of how the similar technique can be used to help recommend third party libraries and API methods to the developer, thereby increasing the efficiency of his development work and thus the software project. This still is an open area of research and clearly brings a lot of value. With the continuous progress in the machine learning fields we can expect further progress in the calculation of similarity between documents and thus automated detection of duplicate bugs.

IMPROVEMENTS

After reading these papers, we have a good understanding of the research and progress in the field of duplicate bug detection. Since the entire progress in this domain is based on the natural language processing technique, we feel that with recent advances in this field there is still a good amount of progress to be made. We also recommend future researchers to evaluate their researches on other large repositories. We also recommend using alternatives to LDA for topic modelling like Explicit Semantic Analysis, to look for improvements in cases when the bug descriptions are very different for the same issue. Also in those cases when less training data is available, the above discussed models do not perform well. In such cases other features can be used like code commit history, identifying buggy components etc.

REFERENCES

[1] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, “Duplicate bug report detection with a combination of information retrieval and topic modeling,” in ASE, 2012.

[2] J. Chang and D. Blei. Relational topic models for document networks. In AIStats, 2009.

[3] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In ICSE’10. ACM, 2010. Tavel, P. 2007. *Modeling and Simulation Design*. AK Peters Ltd., Natick, MA.

[4] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In ASE’11, pages 253–262. IEEE CS, 2011.

[5] Anahita Alipour , Abram Hindle , Eleni Stroulia, A contextual approach towards more accurate duplicate bug report detection, Proceedings of the 10th Working Conference on Mining Software Repositories, May 18-19, 2013, San Francisco, CA, USA

[6] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. ICSM, pages 200-209, 2013.

[7] F. Thung, D. Lo, and J. L. Lawall, "Automated library recommendation. in WCRE, 2013

[8] F. Thung, W. Shaowei, D. Lo, L. Lawall, "Automatic recommendation of API methods from feature requests", 28th International Conference on Automated Software Engineering (ASE‘ 13), pp. 11-15, 2013.

[9] N. Bettenburg, S. Just, A. Schr¨oter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In FSE’08, pages 308–318. ACM, 2008.

[10] D. M. Blei, A. Y. Ng, M. I. Jordan. Latent Dirichlet Allocation. J. Mach. Learn. Res., 3:993–1022, 2003.

[11] V. Guana, F. Rocha, A. Hindle, and E. Stroulia, “Do the stars align? Multidimensional analysis of android’s layered architecture,” in Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE, 2012, pp. 124–127

[12] A. Hindle, N. Ernst, M. Godfrey, and J. Mylopoulos, “Automated topic naming to support cross-project analysis of software maintenance activities,” in Proceedings of the 8th Working Conference on Mining Software Repositories. ACM, 2011, pp. 163–172.

[13] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In ICSE’07. IEEE CS, 2007.