Learning to rank faulty source files for dependent bug reports

Safdari, Nasir, Alrubaye, Hussein, Aljedaani, Wajdi, Baez Baez, Bladimir, DiStasi, Andrew, et al.

Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, Mohamed Wiem Mkaouer, "Learning to rank faulty source files for dependent bug reports," Proc. SPIE 10989, Big Data: Learning, Analytics, and Applications, 109890B (13 May 2019); doi: 10.1117/12.2519226

SPIE.

Event: SPIE Defense + Commercial Sensing, 2019, Baltimore, Maryland, United States

Learning to rank faulty source files for dependent bug reports

Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, Mohamed Wiem Mkaouer

Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester, USA

ABSTRACT

With the rise of autonomous systems, the automation of faults detection and localization becomes critical to their reliability. An automated strategy that can provide a ranked list of faulty modules or files with respect to how likely they contain the root cause of the problem would help in the automation bug localization. Learning from the history if previously located bugs in general, and extracting the dependencies between these bugs in particular, helps in building models to accurately localize any potentially detected bugs. In this study, we propose a novel fault localization solution based on a learning-to-rank strategy, using the history of previously localized bugs and their dependencies as features, to rank files in terms of their likelihood of being a root cause of a bug. The evaluation of our approach has shown its efficiency in localizing dependent bugs.

Keywords: bug localization, machine learning, mining software repositories, software quality

1. INTRODUCTION

Software development teams spend a lot of time fixing bugs on a regular basis. Currently, the process of debugging and correcting defects in a software is a tedious, time consuming, and expensive task. A new mechanism for estimating possible locations in the source code while considering how likely they are to contain the cause of the bug would help developers to reduce their search and increase the productivity. A lot of real-world software projects get an expansive number of bug reports day by day, and addressing them requires much time and effort. Investigating bugs can contribute up to 80% of the aggregate cost for a software project. Therefore, there is a squeezing requirement for automated strategies that makes development teams work easier. This issue has persuaded extensive work proposing automated troubleshooting answers for different cases.

The debugging process summary is explained as when a developer is provided with a bug report and then he/she must replicate the defect and perform several reviews of the source code to find the root of the cause. As software system grows in size, the debugging process becomes challenging as developers typically receive a significantly large number of bug reports, which increases the amount of time needed for their resolution. To cope with this problem, various approaches have designed to automate the bug localization. These approaches mainly rely on determining the similarity between bug reports and different systems characteristics, such as its source code, previous solved bug reports, etc. The similarity between bug reports and these characteristics is calculated using Information-Retrieval (IR) techniques. For a given newly report bug as input, existing bug localization approaches generate as output a list of source files according to their textual similarity to the input bug report. This prunes the search space for debuggers as they start their analysis with a relevant subset of source files instead of the entire codebase. In summary, various techniques have been treating bug localization as a ranking problem for relevant files that are most likely prone to the reported error.

In order to train a model to solve this ranking problem, Learning to Rank approach has been proved to be an effective method.[1–3] This approach has been applied in several domains such as search engines and information retrieval. In this study, a bug report can be observed as a query and the source code files can be searched as a directory of documents. Thus, searching the source code file/files that are known to be the root cause of a bug, can be constructed as a regular structure in Information Retrieval (IR).[4]
Since in this approach, the source code files (directory of documents) can be ranked with respect to how relevant they are to the root cause of a bug, the mentioned approach can be viewed as a ranking problem. In order to

Further author information: (Send correspondence to Mohamed Wiem Mkaouer)
Mohamed Wiem Mkaouer: E-mail: mwmvse@rit.edu

Big Data: Learning, Analytics, and Applications, edited by Fauzia Ahmad,
Proc. of SPIE Vol. 10989, 109890B · © 2019 SPIE · CCC code:
0277-786X/19/$18 · doi: 10.1117/12.2519226

evaluate the relevancy of the bug report to each source code file, the ranking function is constructed based on the combination of weighted features. Although a bug report might have some shared textual similarities with its relevant source code files, mainly there is a vast difference between the natural language used in the bug report and the programming language used in the source code.[5] This lexical gap between the technical syntax used in the programming language and the natural language used in the bug report can be bridged by using the API specification documents that contain the formal explanation of the class and its methods relationships and responsibilities.

Another practical solution to bridge the mentioned lexical gap is to parse the source code files into each method, since in general, the size of a class might be quite large and it can contain numerous methods. out of all those methods, it less often occurs that all of them correspond to the cause of the bug, therefore, this process can help to find the cause in a more precise manner. Furthermore, another domain that can help improve with the lexical similarity between the source code files and bug report is the change history of the source code file. It has been observed that if a source code file is involved with a high number of flaws, it might be the cause of multiple bugs.[6] One crucial implication to take out from the mentioned observation is that if the bug reports that are fixed before the current bug is reported, the textual similarity between the former bug reports and the current bug report describes the association between the files that have been fixed for similar bug reports. in other words, textual similarity between the current bug report and the previously fixed bug reports implies that those file/files that were fixed for the previous bug reports might be the cause of the abnormal behavior described in the current bug report. The mentioned lexical features were all dependent on the textual features of the query (bug report). Besides query-dependent features, to capture the query-independent features, the bug-fixing recency and bug-fixing frequency of each source code file are also calculated. These two features provide information about the file revision history that can help determine the faulty source code files.

By applying the learning-to-rank approach, the weights for each feature can be automatically trained on the bugs that are previously solved. In this process, the previously solved bugs are used as the training set for a specific learning-to-rank algorithm that acts as a ranking function which is able to learn the weights based on different values of the linear combination of features. Unlike a previous study[6] that used a specific version of the source code as the directory of the documents to be searched, by checking out the before fix version of the source code for each bug report, in this study the training data remains more conforming as the search space to locate the root cause of each bug.

Bug reports are provided by users who face abnormal behavior in the software. When a user reports a bug, he/she writes a short explanation of how things went wrong while trying to use specific features of the software. Those bug reports are in natural language format and they must be preprocessed before using them in a learning-to-rank Algorithm. In addition to the bug reports, source code file and API specification documents also need to be preprocessed before they can be used as inputs for the learning-to-rank algorithm. In general, one of the most important tasks in the text preprocessing stage is that words in a corpus or a bag of word need to be reduced to their basic shape. How to process the words to get to their basic shape affects the final results. There are different methods to reduce each word to its basic shape and most common ones are: Stemming and Lemmatizing. There will be more details about the text preprocessing methods in methodology section.

Since learning to rank is a supervised machine learning method, the data needs to be divided into training, validation, and testing sets. there are different approaches to split the dataset into those sets. How to split the dataset intensively affects the performance of the algorithm and consequently the ultimate results in terms of precision and accuracy.

The purpose of this research is to improve the bug localization approach presented by Ye et al.[2] by taking into account the dependency between analyzed bug reports, which constitute the bug fix history. We also evaluate the effect of different natural text preprocessing techniques and also a randomized selection of training folds on the overall performance of learning-to-rank algorithm used in the original study.

The remainder of this thesis is organized as follows: Section 2 provides the necessary background related to bug reports and their localization. Section 3 discusses the related studies. The IR techniques used in this study are detailed in Section 4, while the calculated features are explained in Section 5. Section 6 contains the experimental setting. Experiments are detailed in Section 8. Threats to the validity of our findings are enumerated in Section 9 before concluding in Section 10.

2. BACKGROUND

This section outlines the different activities involved in the bug report life cycle. This process starts when the development team receives the bug report and then the bug is assigned to the specific team member who is more familiar with the connection between the bug report and part of the source code that might be the cause of the abnormal behavior carried out by the bug. Next the assigned team member/members will go through the bug localization process and ultimately find the cause of misbehavior and solve the problem. Figure 2 portrays the life cycle of a bug report.

Bug report: it is a structured record consisting of several attributes that describe a specific anomaly in the source code and how to reproduce it. Typically, a bug report is structured into a bug summary, description, host software, reporter, priority, date of discovery, etc. it is important to note that, unlike source code, bug reports in written natural language.

Duplicate bug report: Two or many different bug reporting the same error in the code. These bug reports may be describing the same failure in the system, or multiple failures, which are originated from the same error in the source code.

Dependent bug report: Two or many different bug reporting different errors, but located in the same code element (class, method). These bug reports may be describing multiple independent failures, but co-located in the source code, which one may block the other in terms of correction. Thus, developers need to be aware of dependent bug to schedule their correction accordingly.

Bug management: In software development, it is crucial to have a repository in which developers can report a particular bug or issue. Not only does it help them keep track of bugs but also it helps to recommend improvements to various bug reports. Since the bug repository is publicly available to anyone, the quality of a particular software project can be enhanced.
Bug triage: It is the process in which bug reports are prioritized and assigned to an appropriate bug fixer. When a bug report is received, several factors are taken into consideration in order to process the request. First, a bug report can be classified based on its importance. If the importance is higher, it will be highly considered to be fixed earlier than the other bugs. Another crucial factor is severity, which also affects its importance. Combined together, it leads to having the bug being prioritized from low to high priority.
Bug localization: Once a bug becomes resolved by the developer, the bug status makes a transition from resolved to be verified. After it is verified, then the bug is defined as closed so that no more occurrences of the bug will be reported.
Bug tracking systems: In software development, developers use a popular platform in which they can report the different types of bugs that a software product can contain. It facilitates the bug tracking, and at the same time help them fix those bugs in an efficient way. There are plenty of recognized bug tracking systems such as Bugzilla, RedMine, JIRA, and others.
Bugzilla: All bug reports, analyzed in this study, are mined from Bugzilla. It is a defect tracking system that allows software developers track and report different types of bugs. Actually, it is a free repository and contains many useful features that it became one of the most popular bug tracking systems used by many developers.
One of the advantages is that Bugzilla can be installed in most operative systems (OS). It runs for Windows, Mac, and Linux. It facilitates the way in which bug fixers choose the appropriate environment for deployment. Another plus is that it is totally free so it prevents any extra cost.
Bugzilla classifies its features between users and administrators. Some essential features include the ability to search for a specific bug in a Google-like search. It helps the user find the bug following certain criteria. For instance, if the developers want to look for the bug which ID is "123456", they can input its ID to retrieve important information such as: (reference: https://bugzilla.readthedocs.io/en/latest/using/editing.html#life-cycle-of-a-bug)

- Status: It shows all the possible statuses of a particular bug. It helps the developer distinguish whether the bug has been fixed or not.

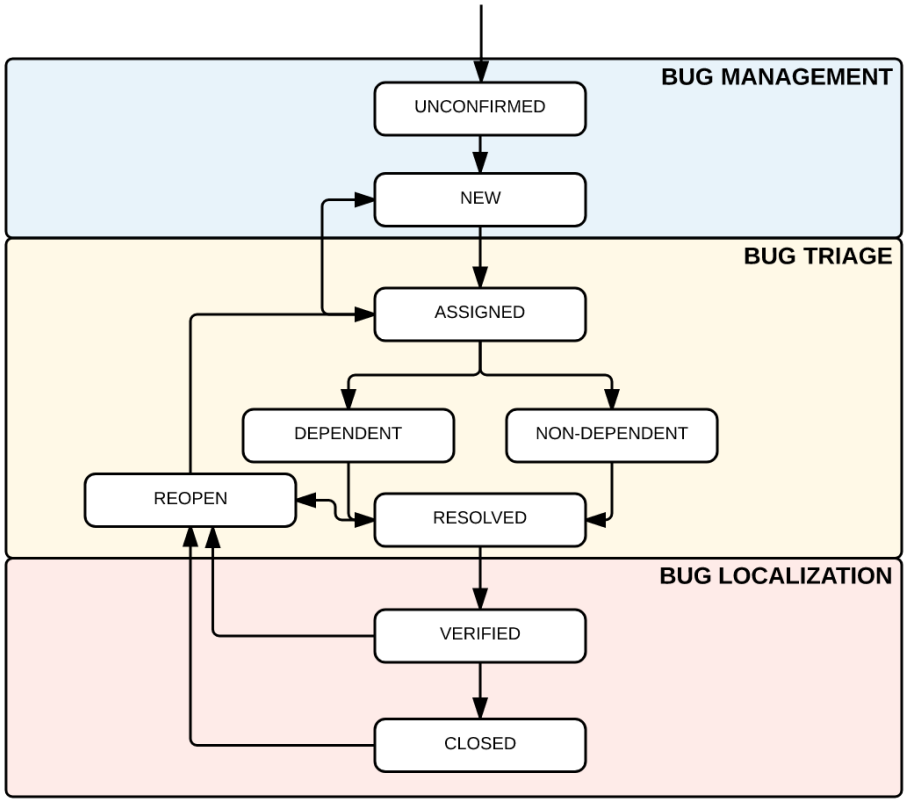

Figure 1. Lifecycle of a bug report.

- Product: Based on the software product. For instance: platform, UI.
- Component: The various sections or modules that compose a software product are defined as components.
- Version: It means that whenever a software product is revised to correct previous errors, developers release a newly modified software.
- Hardware: It is where the bug was originally found.
- Importance: It is based on the grade of severity and priority.
- Target Milestone: Specifies to whom the bug is assigned to.
- Assigned To: As the name states, it is to whom the actual bug was assigned to.
- QA Contact: Here, the person in charge of fixing/posting the bug will be shown here.
- URL: If there is any web link associated with the bug, it will be shown.
- Whiteboard: In this field, developers specify words or terms to make the bug tracker system more organized.
- Keywords. Essential words to identify the characteristics of a particular bug.
- Depends on: The bug which is dependent on another one.
- Blocks: It refers to the bug that blocks another one.

There are different types of status for a bug that are classified as "Open bugs" and "Closed bugs". For open bugs, the first one on the list is *UNCONFIRMED*, which means that there is a bug recently added into the database and it is not marked as valid yet. Once here, the bug state can transition to either *CONFIRMED* or *RESOLVED*. The *CONFIRMED* state refers to a bug which has been validated and it can be *IN_PROGRESS* if a developer is working on it then it becomes *RESOLVED*.
For closed bugs, there is the *RESOLVED* state meaning that a decision has been done. In this state, once it is verified by the QA, the bug can be either given reopened and get some open status or verified and marked as *VERIFIED*. The VERIFIED is the final in the bug status, and it is when the QA has made the appropriate decision to verify the bug.
There is a resolution set depending on the status of a bug. First, the *FIXED* resolution indicates there is a fix for the bug that also has been tested. Next, when the responsible person determines that a bug is not a bug, the resolution will be given to *INVALID*. Another resolution can be also when a bug does not have any possibility to be solved, then it becomes *WONTFIX*. If there is a bug that is duplicate of another, then its resolution will be marked as *DUPLICATE*. Finally, the *WORKSFORME* resolution is given when there is no idea how to fix the particular bug, so if enough information is encountered then the bug could be reopened. Ref. (https://bugs.eclipse.org/bugs/page.cgi?id=fields.html#bug_status)
Bugzilla was developed by Mozilla. It is constantly maintained and updated based on any feedback to the platform. Its key characteristics are that it decreases downtime, increases productivity, improves communication, and elevates the customer satisfaction.
Many popular software projects use Bugzilla as its bug defect tracker system. To mention a few, there are: Eclipse, Linux Kernel, GNOME, KDE, Apache Project, LibreOffice, OpenOffice, and its company itself called Mozilla.

3. RELATED WORK

Bug management:
Other works have studied crash reports as bug report characteristics that help keep track of bugs. Authors in[7] emphasize about correlations in crash reports that can improve bug management. In this work they propose five rules to detect correlated crash types automatically, in which two of them are added to the present work. One rule based on the time crash occurs, and the second one based on the textual similarity of crash types. Also, by using crash correlation groups, they use an algorithm that helps locate and rank any buggy files. In addition, they present a method that identifies bug duplicates. It is said that due to the huge amount of crash reports generated, similar crash reports are grouped together in order to speed up the process because sometimes crash reports may refer to the same bug. They conduct the study on Firefox and Eclipse.
Authors in[8] mention that sometimes there are too many bug reports generated in bug tracking systems from which a considerable percentage pertains to bug duplicates. So, their approach tends to tackle that problem by proposing a method to classify the incoming bug reports into the appropriate category; in this case the ones that are bug duplicates taken into account the important features that a bug report contains for detection. In addition, since several bug reports have contextual similarities such as two reports having the same description but in different words, they propose a formal model to mitigate this issue. They conduct their study using 29,000 bug reports from the Mozilla project.
When detecting bug report duplicates, it is important to use a textual similarity approach in which the descriptions between two bug reports can be compared efficiently. In,[9] authors present three approaches: String-based, Corpus-based, and Knowledge-based. First, for the String-based, the measures are classified into two categories. Character-based and term-based, each one having seven algorithms to be used for analyzing the text. Second, the Corpus-based similarity, which is a semantic similarity having six measures. It analyzes the text based on a context from words already stored in a large dataset. Third, Knowledge-based similarity which focuses on the degree of the semantic link among words and has five measures. This one is grouped into two categories: similarity, and relatedness. In this case, the semantic context is taken from semantic networks.
In this work,[10] the authors point out that a huge amount of crash reports are generated on a daily basis which makes it difficult to manage. To mitigate that problem, they propose a two-level group approach in which crash reports that are linked to the same bug are grouped together. By doing so, it facilitates the triage process and reduces the time of bug fixing. However, if a crash report refers to multiple bugs, it would become a tedious

process. So for each crash report, the stack traces are helpful to identify the bug to see whether is a duplicate or not. In order to evaluate the similarity between stack traces of the crash reports, they used the Levenshtein distance. They conduct an empirical study on ten Firefox releases. It is mentioned that their approach helped decrease bug fixing time by more than 5%.

Similar to research that has been conducted before, authors in[11] present an approach to decrease the number of bug report duplicates by extending the BM25F formula for larger queries. In the study, they increase the accuracy improvement by 10-27% in the recall rate. Not only do they take into consideration the textual features, but also category features as well in order to enhance the accuracy of the results. They studied the bug repositories from Mozilla, Eclipse, and OpenOffice.

Bug triage:

Other researchers that have conducted studies on bug triage[12] focus on historical bug-fix information that means that each developer that have fixed bugs previously, have a generated historical information that contains all his/her activities of fixing bugs.

Some bug triage methods are effective when team size is moderate but they may not be effective when the project size increases. In order to solve that issue, the authors propose an automated bug triage tool named BugFixer that facilitates the bug fixing process for larger projects.

They conducted the study on three large-scale software projects (Eclipse, Mozilla, Netbeans) plus two smaller projects. Based on the outcome, it shows that Bugfixer performs better when having larger projects. So, traditional approaches have revealed that when it happens, their methods do not achieve the expectations. It is mentioned that developer expertise cannot be learned from historical data. In some cases the tool performance may not be efficient as expected since it may fail to provide good recommendation for some bug reports.

Similar studies suggest a semi-automated approach to speed up the process of bug triage when new bug reports are assigned to developers. Authors in[13] use machine learning to rank a list of potential developers that can fix a bug. They used Bugzilla as the main bug repository that includes two projects: Eclipse, and Firefox. To validate their approach, they used a third project called GCC but the results were not promising since the outcome is lower than the expected.

Bug localization:

Authors in[14] explain that when a software product is released, it is usual that several bugs appear. Also, it is mentioned that in most cases developers submit bug reports that later are detected as duplicates. Thus, in order to speed up the bug duplicate detection, they developed a tool using NLP that helps in the detection of duplicates. For the tool, there are various techniques used in the detection process such as tokenizing, stemming, and stop words removal. However, it is stated that when the data that to be processed is very large, the operation could be complex. One way to avoid that is to filter out data that is not important and have only the data that is relevant. In addition, when using an NLP algorithm the bug reports are ranked based on their similarity so that only the high-ranked are shown. For their case study, they analyzed defect reports at Sony Erickson Mobile Communications using two approaches. In the first approach, they chose bug reports that were classified as duplicates in each report using batch runs. In the second approach, they conducted interviews with testers and analysts in order to evaluate their tool. It is concluded that the tool is very useful even though only 40% of duplicates can be detected.

There is another approach when locating bugs. Authors in[15] propose an idea to locate bugs at a code change level. It means that whenever there is a modification within the code, developers can identify those changes that can help find the bug. They proposed a tool named Locus which provides a better understanding of how bugs can be fixed faster than the traditional approaches. Three ranking models are taken into consideration such as the NL (Natural Language) Model, CE (Code Entity) Model, and Boosting Model. For the NL, tokens are extracted from bug reports in which only the summary and description sections are considered. Meanwhile in the CE, which refers to the code entity, components such as package names, class names, and methods names are treated as single tokens before preprocessing. Now for the boosting model, they used the algorithm used by Google and adapted it according to the study. In this model, the goal is to find the highest suspiciousness score that leads to a file to be buggy.

Similarly, other authors[6] point out that locating source code that needs to be modified can be tedious. For this reason, they propose a method named "BugLocator" which is used to locate targeted buggy-files. It is an automated process that reduces waiting time and cost, and increases user satisfaction. According to their study,

the method outperforms better than other suggested approaches by using a large dataset of bugs from four open source projects (Eclipse, AspectJ, SWT, and ZXing).
Many research works have been focused on either Information-Retrieval (IR) or Machine Learning (ML) techniques for bug localization. Authors in,[16] instead, propose a combined approach to detect the most likely buggy files in projects which is advanced IR technique with Deep Neural Network (DNN). They evaluated their idea with real-word projects over existing bug localization approaches. It is showed that their technique ranks higher among other similar studies. DNN also addresses lexical mismatch that the other IR techniques contain.
Other authors[17] propose a tool named BLUiR (Bug Localization using Structured Information Retrieval), which improves the bug localization techniques. As shown in the results, BLUiR outperforms better than any other tools. First, it takes a bug report as a query, similar to other approaches, and then extract the summary and description that later is tokenized. Those tokens are converted into queries and are processed for the structured retrieval. Next, the source code files are treated as documents, which later the Abstract Struct Tree is applied; elements such as classes, methods, variables, comments are tokenized and converted to structured documents that later are indexed then sent to the Structured Retrieval.

4. INFORMATION RETRIEVAL

before we introduce the features extracted from the domain knowledge, we will outline the necessary background of the IR techniques used in the extraction process.

4.1 Vector Space Model

The classic Vector Space Model (VSM) can be applied to convert the text into vectors of term weights. The reason to convert the text into vectors is that, mainly machine learning algorithms accept inputs in the form of matrices. We can take the bug report as a query and source code files as the directory of the documents to be searched for, then VSM can be used to convert the text of the bug report and source code files into vectors. Those vectors of term weights can be represented as one-row matrix representing the textual features of each bug report or source code file. In any document d (bug report or source code file), the term weights $w_{t,d}$ of each term t contained in d is calculated as:

$$w_{t,d} = \mathrm{TF}_{t,d} \times \mathrm{IDF}_t \tag{1}$$

In the above equation, $TF_{t,d}$ is the number of times term t appears in document d. IDF_t is the inverse document frequency $= log\frac{N}{DF_t}$. TF-IDF is one of the widely used methods to convert text into vectors of term weights in information retrieval.[4]

4.2 Text Preprocessing Methods and Cosine Similarity

A similar methodology explained in[18] was applied for text pre-processing. In order for the bug reports and source code files to be converted into vectors of term weights, first they need to be preprocessed and cleaned; put into a format that the machine learning algorithms will accept. The first step is to tokenize each body of text by splitting it into its constituent set of words. After that, punctuation and stop words are removed since they do not play any role as features for the learning-to-rank algorithm. Next, all the words are converted to lower case and then stemmed using the Porter Stemmer in the NLTK package. The goal of stemming is to reduce the number of inflectional forms of words appearing in the bug reports or source code files; it will cause words such as "performance" and "performing" to syntactically match one another by reducing them to their base word– "perform". This helps decrease the size of the vocabulary space and improve the volume of the feature space in the corpus.
In addition to stemming, another method that is used to reduce the number of inflectional words in bug reports and source code files is Lemmatization. In Lemmatization, the part of the speech of a word should be first determined and the normalization rules will be different for different part of speech. In this process, the inflections are chopped off based on defined rules and it relies on some lexical knowledge base to provide the correct basic

Bug ID: 30949
Summary: After Failed Include, Request and Response not Unwrapped.
Descrption:
In org.apache.catalina.core.ApplicationDispatcher.doInclude(ServletRequest request, ServletResponse response), after invoke() is called to perform the include, the request and response objects are normally unwrapped. However, if a ServletException or IOException is thrown, the unwrapping does not take place. This leads to problems in environments in which cross context includes are being performed. For example, 1. Web App A performs an include to a Servlet in Web App B 2. The Servlet in Web App B throws a ServletException 3. Web App A catches the ServletException and attempts to forward to an error jsp. Step 3 above will fail due to the fact that the request has not been unwrapped. The active request will contain the ServletContext of web app B and the jsp file will not be found.

Figure 2. Tomcat Bug Report Number 30949.

form of words. while in stemming, it works like a sharp knife sometimes chopping off too little or too much from the words. For instance, the plural word geese changes to goose and words meanness and meaning remain untouched while stemming convert both of them to "mean". Finally, each corpus is transformed into vector space model (VSM) using the tf-idf vectorizer in Python's SKlearn package to extract the textual features.
After converting the text in a bug report r or a source code file s into vectors of term weights, the standard *cosine similarity* method can be used to calculate the textual similarity between a bug report and a source code file using their vectors of term weights.

$$CosineSim(r,s) = \frac{\overrightarrow{r}.\overrightarrow{s}}{||\overrightarrow{r}|| \quad ||\overrightarrow{s}||} \tag{2}$$

Equation 5.3 calculates how similar two documents are in the vector space by multiplying the dot product of those two vectors and divide it by their magnitude. It is important to note, this is a measure of orientation and not the magnitude. Here, the magnitude of each vector of term weights in the document d is not the only element in calculating the similarity. The angle between two vectors determines the similarity of the two vectors. For instance, if we have a vector that is quite long and another vector that is shorter and the angle between those vectors are small, then Cosine similarity tends to ignore the high length and calculates the measurement based on the angle between those two vectors. In practice, if we have a document that contains the word "book" 100 times and another document that contains the same word 25 times, the Euclidean distance between the vector representation of term weights of those two document will be higher but the angle will still be small since they are pointing to the same direction, and that determines the similarity score when we are comparing documents.

5. FEATURE ENGINEERING

This section will discuss how features are extracted from the specific relationship of each bug report with every single source code file in a software project. It is important to note that in this study, a total of six different features that were discussed in the original paper[2] extracted based on the domain knowledge understanding of how a bug report can be related to a source code file. Out of those six features, the first three of them are using the *CosineSim(r,s)* to measure some lexical similarity.

5.1 Source Code Files Lexical Similarity

Measuring the lexical similarity between the bug report and a source code file can help understand the direct relationship of the textual features between the bug report and source code files. It often occurs that the size of source code is large; it contains multiple methods and different code blocks. While measuring the lexical similarity between a bug report and source code file, it is a fact that the root cause for a bug is placed in one or few methods and not the whole class. As a result, when the source code file is large, the length of its VSM representation of term weights will be longer in compare to the same vector for the bug report. This can result in a small cosine similarity score between the large file and the bug report. This problem can be addressed by segmenting the source code file into each method and then calculate the cosine similarity scores for each method

Interface Member
The Member interface, defines a member in the group. Each member can carry a set of properties, defined by the actual implementation. A member is identified by the host/ip/uniqueId The host is what interface the member is listening to, to receive data The port is what port the member is listening to, to receive data The uniqueId defines the session id for the member. This is an important feature since a member that has crashed and the starts up again on the same port/host is not guaranteed to be the same member, so no state transfers will ever be confused.

Figure 3. API specification for Member Interface from Tomcat project

and the whole class. after that we can take the maximum of all those scores. This way, we can make sure that the lexical similarity is calculated for each method and the whole class and the final score is the maximum of those calculated scores. The VSM representation of a bug report contains its both summary and description. the summary of a bug report includes a brief description of the abnormal behavior experienced by the user. The description part is a more detailed explanation of what went wrong when the user experienced the abnormal behavior in the software. Figure 2 shows a sample bug report from Tomcat project.

$$\Phi_1(r,s) = \max(\{CosineSim(r,s)\} \cup \{CosineSim(r,m) | m \in s\}) \tag{3}$$

As shown in this equation, cosine similarity score is calculated for each method m in the file s.

5.2 API Specifications Lexical Similarity

One way to improve the lexical similarity between two documents is to make sure that both them are constructed in a similar format. for instance, source code is written in programming languages like Java. In case the source code is not well documented by appropriate comments, then the cosine similarity may not be able to direct us to the root cause of an abnormal behavior explained in a bug report. Since bug reports are generally written in natural language format (e.g. English), they are not in the same format as programming languages. Bug reports sometimes contain some snippets of the source code, or the name of a class or the name of a method but often, they explain the problem with the software in natural language format and they do not provide relevant information that can be found by measuring the textual similarity between the bug report and source code file. As a result, we can see that the source code files lexical similarity explained in the previous section can provide a good similarity score if those files come with enough comments or the bug report provide enough source code snippets that correspond to the root cause files.

This lexical gap can be bridged by using the API specification documents of the software project. It is a common practice for the development teams to write API specification notes to keep track of different classes, super classes, interfaces, and their relationships. The API specification documents can help the current and future development team members to understand how different components are related and working together in the software project. The API specifications are written in natural language, theretofore, they are in the same format as bug reports and it can help bridge the lexical gap between the programming language of source code files and natural language of bug reports. Fig. 3 is an example of an API specification for Member interface in Tomcat project. This feature can be computed as follows:

$$\Phi_2(r,s) = CosineSim(r, s.api) \tag{4}$$

5.3 Collaborative Filtering Score

In collaborative filtering, the lexical similarity score between the bug reports can be used to find the corresponding source code file/files for a newly reported bug report. It has been observed[19] that similar bug reports correspond to the same file/files during the bug fixing process. This technique, collaborative filtering, has been used to improve the accuracy of the recommender systems.[20] As a result, this approach can be used in our case which is an information retrieval problem. For any given bug report r and a source code file s, the set of all bug reports that correspond specifically to file s and were fixed before r was reported, can be called *set(r,s)*. We can compute the collaborative filtering score as follows:

$$\Phi_3(r, s) = CosineSim(r, set(r, s)) \tag{5}$$

5.4 Collaborative Dependency Score

This feature is similar to the collaborative filtering score as it also uses the previously fixed bug reports, but it focuses on detecting whether the most similar bug reports contain any dependent bug reports, and if they do, the set of dependent bug reports' corrected files are also considered suspicious. This feature reveals the hidden impact of any dependency between bug reports since they share the same root cause, and so, it bridges the gap between lexical similarity, since dependent bugs may not necessarily have a strong similarity with the input bug report.

For any given bug report r and a source code file s, the set of all bug reports that correspond specifically to file s and were fixed before r was reported, can be called *set(r,s)*, and their corresponding dependent bug reports, which are called *dep(r), and contain the fixed files s'*. We can compute the collaborative dependency score as follows:

$$\Phi_3(r, s') = CosineSim(r, set(dep(r), s)) \tag{6}$$

5.5 Class Name Similarity

Sometimes the bug in software is captured by the professional users and when they provide the report, they trace the abnormal behavior and provide some extra technical details. Within the extra details, sometimes they mention the name of the class that might be relevant to the root cause of the bug. As mentioned in the original study,[2] the hypothesis is that the length of the class name mentioned in the bug report is relative to how strong this feature represent its value. In other words, when the class name is longer and more specific, the feature must provide more accurate clues about the corresponding source code files. Therefore, this feature is evaluated by the length of the class name mentioned in the report and it is computed as follows:

$$\Phi_4(r, s) = \begin{cases} |s.class|, & \text{if } s.class \in r \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

It is important to note that the value of the first three features mentioned in this study are between 0 and 1 since the cosine similarity function always returns a number between 0 and 1 depending on how similar two documents are; 0 means not similar at all and 1 means both documents are identical. But the length of the class name in this case can be variable. In order to create a dataset that is scaled uniformly, all the values need be in the same range, i.e. ([0,1]). As a result, all the features will be normalized automatically. more details about the normalization process will be provided at the end of this section.

5.6 File Alteration History

It has been observed that by evaluating the source code alteration history, we can achieve information that can help detect and predict files that are more prone to cause bugs.[21] For instance, a source code file that has been fixed several times recently is more likely to correspond to newly reported bugs.

5.6.1 Bug Fixing Recency

This feature applies this notion that if a source code file was fixed recently, it means that it is more likely to contain more bugs. For a random bug report r, $r.month$ is the month that bug report r was created. As explained in section 5.3, *set(r,s)* is the set of all bug reports that correspond specifically to file s and were fixed before r was reported. The most recent previously fixed bug report in the *set(r,s)* can be called *latest(r,s)*. This feature is calculated as follows:

$$\Phi_5(r,s) = (r.month - latest(r,s).month + 1)^{-1} \tag{8}$$

This equation shows that bug fixing recency is implemented as the inverse of distance in months between bug report r and $latest(r,s)$. In other words, the value of this feature is 1 if file s was fixed in the same month as bug report r was reported and it is 0.5 if file s was fixed one month before r was reported.

5.6.2 Bug Fixing Frequency

As described in section 5.3, *set(r,s)* is the set of all bug reports that correspond specifically to file s and were fixed before r was reported. This feature is defined simply as the number of times a source code file has been fixed before the current bug report being reported.

$$\Phi_6(r,s) = |set(r,s)| \tag{9}$$

The value of this feature will also be quite variable. Therefore, it will be automatically normalized as described in the next part.

5.7 Feature Values Normalization

Most of machine learning algorithms tend to perform better when the values in the dataset are not in different extreme ranges. Comparing values that are not on the same scale leads to bad results in a machine learning model. In this case, the first three features have values between 0 and 1. In order to normalize values in the dataset, all features need to be in the same range. For any arbitrary feature in the training dataset, let $\Phi.min$ be the minimum observed value and $\Phi.max$ be the maximum observed value. While analyzing the data from the testing set, there might be feature values that are larger than $\Phi.max$ or smaller than $\Phi.min$ and all data in the training and testing set will be normalized as follows:

$$\Phi_Scaled = \begin{cases} 0, & \text{if } \Phi < \Phi.min \\ \frac{\Phi - \Phi.min}{\Phi.max - \Phi.min}, & \text{if } \Phi.min \leq \Phi \leq \Phi.max \\ 1, & \text{if } \Phi > \Phi.max \end{cases} \tag{10}$$

5.8 Ranking Model

Learning-to-rank is a category of supervised machine learning algorithms that are used to solve ranking problems in information retrieval.[22] In general, there are two stages of learning-to-rank: first is the learning stage and second is the deployment (testing) stage. This method works at the first stage by preparing the training set. more details on how to prepare the training set are explained in section 6. The result of the learning stage are the weights for the features that the algorithm learns to optimize. In general, the learning-to-rank algorithm tries to construct a model to minimize the cost (error) by fitting weights for the features that result in optimized ranking. At the deployment stage, for each bug report (query), the learned weights are applied to the features

and each source code file (document) receives a score based on its feature values and then the list of the source code files are sorted based on their scores.

$$f(r,s) = \mathrm{w}^T \Phi(r,s) = \sum_{i=1}^{k} w_i * \Phi_i(r,s) \tag{11}$$

As formulated by equation 5.1,[2] for each bug report r and any source code file s in the before-fix version of the source code, *f(r,s)* is a function that provides the final score based on the weighted sum of k features. $\Phi_i(r,s)$ calculates the individual score based on the relationship between r and s for each feature. The most important component in the equation are parameters w_i that are trained by a learning-to-rank algorithm on previously solved bug reports.

6. EXPERIMENTAL SETTING

6.1 Dataset Collection

For this study, only bug reports that were in *resolved fixed*, *verified fixed*, or *closed fixed* status from Tomcat project were used. Tomcat is a widely used open source web application server and servlet container. Collecting the bug reports for Tomcat project was done using Bugzilla. It is a tool for issue tracking activities and user can provide their inputs using Bugzilla while facing an abnormal behavior in the software. Moreover, Git is the primary version control system for Tomcat project.
As mentioned briefly in the Introduction section, previous studies used a fixed code revision for their Bug Localization approaches. But it is a fact that bugs are found in different versions of the source code. Using the fixed version of the source code can affect the performance of the Bug Localization systems. Some of the problems using the fixed version of the source code include: the future versions of a software can have important bug-fixing information for older bugs. Furthermore, a file that corresponds to the root cause of a bug may not even exist in the fixed version of the source code. For Tomcat project in this study, a total number of 1054 bug reports are used for training and testing sets. The time range of the reported bugs is from early 2002 to early 2014. The maximum, median, and minimum number of files fixed per bug report are 94, 1, and 1 respectively. The median number of Java files in difference versions of the project source code is 1552 an the number of API specifications are 389.[2] It is important to note that the exact version of the source code for which the bugs were reported is not available all the times. As a result, the version of the source code that was committed right before the fix was used in this study. It is most reasonable to use the mentioned revision of the source code since the relevant fix had not been pushed in the repository and the bug still existed in the version.

6.2 Training Steps

As explained in the Ranking Model section, the ranking function *f(r,s)*, equation (1), is a function that provides the final score based on the weighted sum of k features. $\Phi_i(r,s)$ calculates the individual score based on the relationship between a given bug report r and each source code file s for each feature.

$$f(r,s) = \mathrm{w}^T \Phi(r,s) = \sum_{i=1}^{k} w_i * \Phi_i(r,s) \tag{12}$$

Each model parameter w_i is trained using the learning-to-rank approach. The SVM rank package[23] was suggested as an optimized implementation of the approach by the original paper.[2] The SVM rank basically aims to solve an optimization problem. Here, the optimization problem is to find values for w_i such that the relevant files for each bug are ranked at the top of the list. In other words, if file p is relevant to be the root cause of the problem for bug report r and file n is not relevant to be the root cause of the problem for the same bug, then the objective of the optimization approach is to find parameters w_i for each feature such that *f(p,r) > f(n,r)*.

6.3 Tuning the Capacity Parameters

Choosing the size of training set determines the performance of SVM rank. As mentioned in the previous section, the number of relevant files for a bug is very small; The median of relevant files is 1. Therefore, the size of the training set is affected by the number of samples in the set and the number of irrelevant files used for each bug report. On the other hand, note that the number of irrelevant files is also very large; The median number of Java files in different versions of the project source code is 1552. Using all the irrelevant files makes the training time unfeasible in terms of time complexity. After evaluating different number of samples for the training set using the Mean Average Precision (MAP) and Accuracy@k metrics, using the top 250 irrelevant files that have the highest Cosine similarity score with the bug report r resulted in the optimal range of performance for Tomcat project. More details about the evaluation metrics will be provided in the next section.
Another factor that plays an important role in the ultimate performance of the system is choosing the optimal value for C parameter in the SVM rank algorithm. For the general SVM algorithm, choosing larger values for C, the optimization process chooses a smaller margin hyperplane if that hyperplane performs better getting all the training samples classified correctly and vice versa for smaller values for C. Basically in SVM algorithm, there is no specific rule that provides guidance on choosing the optimal value for C parameter. The C parameter dominates the optimization process in the SVM algorithm on how much misclassification rate can be avoided in the training sample. However, the C value in SVM rank is correlated with the number of samples in the training set in general. An effective way to make sure that the optimal value for C parameter is chosen is using the Gridsearch. In this approach, different C values are tested to achieve the highest performance.

6.4 Training Methods

One training method that is used in the original paper[2] suggests that the bug reports must be sorted based on their reported time in chronological order. After that, the dataset is split into k equal size folds, $fold_1, fold_2, ..., fold_k$; In this study, k is 10. The folds are organized as $fold_1$ having the oldest bug reports and $fold_{10}$ having the most recent bug reports. As mentioned before, for this study, 1054 bug reports from Tomcat project creates the primary dataset, therefore, each fold will contain 105 samples. The intuition here is that if the ranking model is trained on $fold_k$ and tested on the previous fold, $fold_{k-1}$, it will have a better performance. Thus, the ranking model is always trained on the most recent bug reports.
In addition to the mentioned training approach, choosing the training folds randomly is also compared to evaluate the role of randomized selection of samples on the overall performance of the ranking model. In related studies,[24,25] the authors applied the randomized selection of samples for the training set and they were able to improve the results of the trained model relatively. The main objective of applying the randomized selection of samples for the training set is to check if the intuition explained in the original paper on creating disjoint training folds based on the sorted chronological order can really improve the overall performance of the ranking model.
The purpose of training is to attain the weights for each feature in a specific fold. Testing a fold works with multiplying the attained weights by the values of the features on the target fold. In other words, the value of each feature in the target fold will be multiplied by the attained weight of that feature from the training fold. Finally, all the resulted values must be added together for each file s and each bug report r.

7. EVALUATION METRICS

The ranking model explained in the Ranking Model section, computes the weighted scoring function $f(r, s)$, for any given bug report r and each source code file s. This process provides a list of ranked files for each bug report, called the system ranking. I order to evaluate the results, the system ranking is compared with the actual ranking in which the relevant files are at the top of the list. The metrics that are used to evaluate the overall performance of the proposed approach are described as:

1. *Accuracy@k*: this metric describes the accuracy in the percentage of the ranking model when it makes at least one correct recommendation in the top k ranked files for the bug reports.

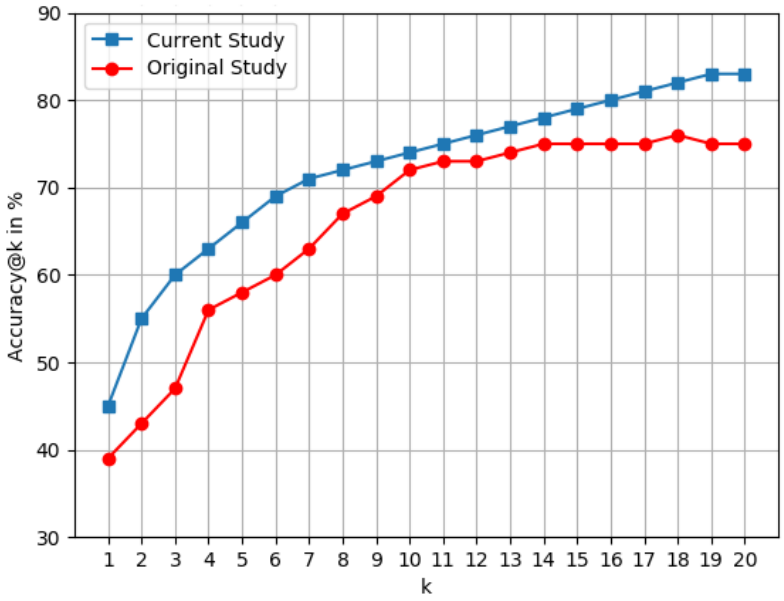

Figure 4. Accuracy@k graph comparing the original study and the current one.

2. *Mean Average Precision (MAP)*: this metric is widely used in evaluating different classifiers and also in information retrieval.[4] *MAP* measures the quality of the information retrieval approach when a query might have several relevant documents.

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|}, \quad AvgP = \frac{Prec@k}{|K|} \tag{13}$$

$$Prec@k = \frac{number \quad of \quad relevant \quad docs \quad topk}{k} \tag{14}$$

In equation 12, Q is the set of all queries; In this case all the bug reports. Average precision (AvgP), is the Prec@k divided by K; K is the set of the positions of the relevant documents in the ultimate ranked list. Prec@k, equation 13, is the overall precision divided by the k top documents in the ultimate ranked list.

8. EXPERIMENTAL RESULTS

One of the objectives of the experiments in this study is to extend the bug localization approach explained in the original paper.[2] It is very important to understand how domain knowledge can help build features for the proper machine learning algorithm. It is a sophisticated process to analyze and discover how different textual similarities and file revision history between the bug report and specific components of a software project can help find the relevant source code files for a bug. Feature engineering in this study describes the process in which raw data turns into useful information that can be used as effective inputs for the proper machine learning algorithm. Another objective of this experiment is to understand whether taking into account the dependency between file can help in improving the accuracy of the original learning-to-rank, when the input bug report actually depends on existing bug reports. Therefore, we compare between the original learning-to-rank along with the one augmented with the newly introduced feature, when given dependent bug reports. To do so, we define the following research question.

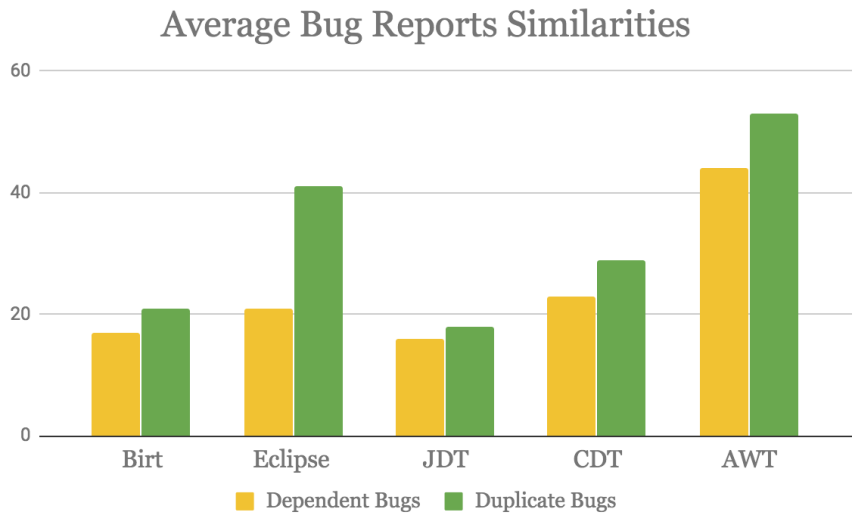

Figure 5. Average bug report similarities for duplicates and dependents.

RQ1: Does augmented learning-to-rank model outperforms the original one when handling bug reports exhibiting potential dependencies with existing bug reports?

One important point to note is that the authors in the original paper implemented this bug localization approach in Java language. they ran the experiment on six large-scale open-source Java projects; Those projects are: Eclipse UI, Tomcat, AspectJ, SWT, Birt, and JDT. However, in this study the experiments are implemented in Python and because of the time constraints, only Tomcat project was analyzed. Since the target projects are all Java based, parsing the source code of those projects using Python had some limitations. The authors in the original study applied the AST parser which is a tool designed to parse merely Java source code, to segment each method from the Java files. On the other hand, parsing the Java source code in Python was done using a library called Plyj; This library has some limitations in terms of detecting specific methods native to Java. As a result, the performance of the ranking model was slightly affected by the mentioned limitation in this study.

As it is shown in Figure 4, the introduction of a feature considering the dependency between bug reports has scored a slight improvement of the current results compared with the original study. As seen in the figure, the *Accuracy@k* is on average better for the current learning-to-rank, for k varying from 1 to 20. Moreover, the MAP was increased by 0.5 from what the authors achieved previously. Thus the MAP for Tomcat project achieved in this study is 0.54. In order to provide a better comparison between the studies, Table 1 provides the average model parameters, w_i for six different trained weights averaged over 9 folds. The value for each weight represents the importance of it in the ranking model and it is clear that the first three features have the highest weights and therefore they have a higher significance.

Table 1. Comparison of average model parameters.

Tomcat Project	w_1	w_2	w_3	w_4	w_5	w_6
Original Study	14.00	3.69	6.38	1.45	0.64	1.67
Current Study	19.23	2.87	4.68	0.97	0.84	1.87

To better understand the difference between dependent and duplicate bug reports, from the lexical similarity perspective, we calculate the average lexical similarities of both dependent and duplicate bug reports. Figure 5 details our findings.

As illustrated in Figure 5, the textual similarity between bug reports helps in better detecting duplicates. Contrary to the dependent bugs which may have very low textual similarity compared to duplicate bugs.

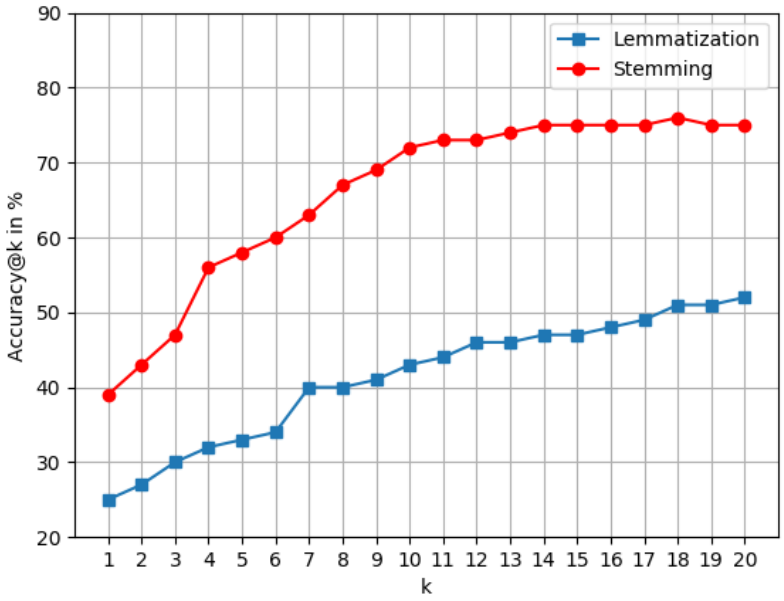

Figure 6. Accuracy@k graph comparing Lemmatization vs. Stemming.

In addition to replication of the bug localization approach, the other objective of the experiments in this study is to address the following research questions:

RQ2: Does Lemmatization of the documents (i.e. bug reports and source code files) make a positive effect on the cosine similarity score and consequently improve the overall performance of the ranking model?
In section 4, it was explained that in the Lemmatization process, the inflections of a word in a document are chopped off based on specific rules and it depends on some lexical knowledge-base to provide correct basic form of words. However, in terms of computing the Cosine similarity between the bug report and source code file, Lemmatization is not able to improve the Accuracy@k and MAP metrics and the overall performance of the ranking model is also affected by this. In Figure 6. we can see how Lemmatization affected the results of the ranking model in terms of Accuracy@k metric and also the MAP achieved using Lemmatization is 0.23. This collapse in the results shows that Lemmatization is not able to improve the cosine similarity between documents with different formats (i.e. source code and bug report).

RQ3: Does the randomized selection of samples for the training folds improve the overall performance of the ranking model?
To answer this question, instead of sorting the dataset and create folds in a chronological order , 10 random folds were selected and the training was done in the order that the folds were created. Thus, the training was done on the first fold and the learned weights were applied on the next fold all the way through the last fold. As it is shown in Figure 7, training on random folds drastically affects the accuracy of the results. The Accuracy@k is dropped to 54% for top 20 relevant documents. This low performance of random folds implies that training on folds that are chronologically sorted have a positive effect on the overall performance of the ranking model. When the ranking model is always trained on the most recent bug reports, the learned weights better match the properties of the testing fold and it improves the overall accuracy.

9. THREATS TO VALIDITY

There are threats to the validity of this study that we elaborate as follows:

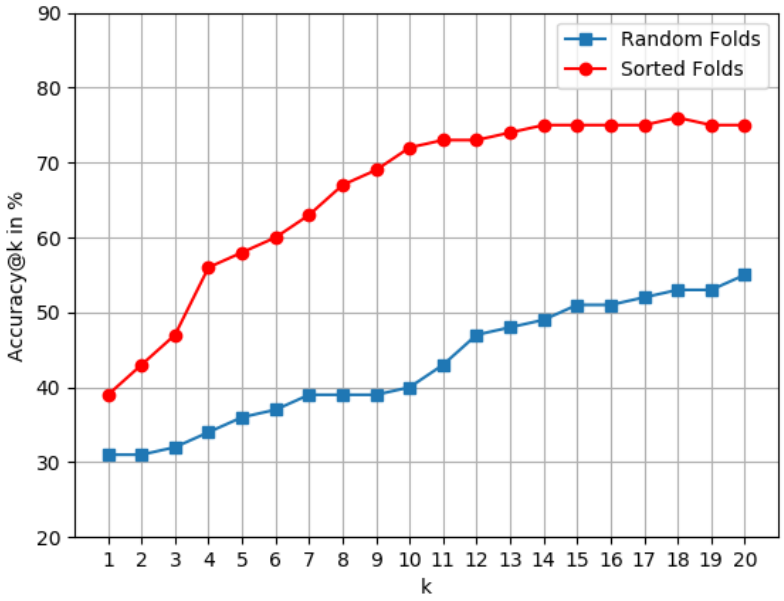

Figure 7. Accuracy@k graph comparing training on Random vs. Sorted folds.

- The experiments in this study and the original paper were all ran on open source Java-based projects. Sometimes the software development practices in open source projects are different from those developed by well-managed software companies. As future work for this study, commercial projects can be evaluated to check the validity of this approach.
- This approach relies on good documenting practices; The source code needs to be well documented with proper comments and the project must have enough API specification documents.
- Similar to the previous case, the bug reports need to provide important details that can help the developers locate the cause of a bug in the project. If the bug report is too short or does not include necessary information, then it will significantly affect the performance of the ranking model.
- Tuning the capacity parameters for machine learning algorithms can vary based on the structure of the dataset and the problem that needs to be solved. For instance, SVM rank in this study has several parameters that work according to the input data and the type of the problem. It is often hard to say that one algorithm works best for a specific problem.

10. CONCLUSION

Once the development team receives a new bug, they need to find which file/files are the root cause of the abnormal behavior. When the size of the project is large, developers need to examine large number of files in order to locate the relevant file/files corresponding to the bug. This process is often a tedious, costly, and time consuming task. In this study, we improved the automated bug localization technique[2] for the detectio of bug reports which are potentially dependent to existing bug reports. The ranking model is trained on the previously solved bug reports by evaluating the relationship between bug report and different elements of a software project such as: textual features of the source code and bug report itself, API specification documents, and file revision history. In addition, we extended the research by evaluating different text preprocessing techniques like, Stemming and Lemmatization and also randomized selection of the training folds on the overall performance of the learning-to-rank algorithm. The experiments show that our newly added feature improves the performance of

detecting faulty files. Also, we show that lemmatization and randomized selection of the training folds results in lower accuracy and precision in ranking the relevant files for a given bug.

As part of our future work, we want to extend our analysis by incorporating more projects. We also want to extend our feature space to better capture the domain knowledge, along with addressing few of the limitations of our current study that we enumerated in the threats to validity.

ACKNOWLEDGMENTS

We would like to thank the authors of the original study[2] for providing us with their dataset and answering questions about how to reproduce their approach.

REFERENCES

[1] Tian, Y., Wijedasa, D., Lo, D., and Le Goues, C., "Learning to rank for bug report assignee recommendation," in [*Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*], 1–10, IEEE (2016).

[2] Ye, X., Bunescu, R., and Liu, C., "Learning to rank relevant files for bug reports using domain knowledge," in [*Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*], 689–699, ACM (2014).

[3] B Le, T.-D., Lo, D., Le Goues, C., and Grunske, L., "A learning-to-rank based fault localization approach using likely invariants," in [*Proceedings of the 25th International Symposium on Software Testing and Analysis*], 177–188, ACM (2016).

[4] Manning, C. D., Raghavan, P., Schütze, H., et al., [*Introduction to information retrieval*], vol. 1, Cambridge university press Cambridge (2008).

[5] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T., "What makes a good bug report?," in [*Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*], 308–318, ACM (2008).

[6] Zhou, J., Zhang, H., and Lo, D., "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in [*Proceedings of the 34th International Conference on Software Engineering*], 14–24, IEEE Press (2012).

[7] Wang, S., Khomh, F., and Zou, Y., "Improving bug management using correlations in crash reports," *Empirical Software Engineering* **21**(2), 337–367 (2016).

[8] Jalbert, N. and Weimer, W., "Automated duplicate detection for bug tracking systems," in [*Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*], 52–61, IEEE (2008).

[9] Gomaa, W. H. and Fahmy, A. A., "A survey of text similarity approaches," *International Journal of Computer Applications* **68**(13) (2013).

[10] Dhaliwal, T., Khomh, F., and Zou, Y., "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in [*Software Maintenance (ICSM), 2011 27th IEEE International Conference on*], 333–342, IEEE (2011).

[11] Sun, C., Lo, D., Khoo, S.-C., and Jiang, J., "Towards more accurate retrieval of duplicate bug reports," in [*Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*], 253–262, IEEE Computer Society (2011).

[12] Hu, H., Zhang, H., Xuan, J., and Sun, W., "Effective bug triage based on historical bug-fix information," in [*Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*], 122–132, IEEE (2014).

[13] Anvik, J., Hiew, L., and Murphy, G. C., "Who should fix this bug?," in [*Proceedings of the 28th international conference on Software engineering*], 361–370, ACM (2006).

[14] Runeson, P., Alexandersson, M., and Nyholm, O., "Detection of duplicate defect reports using natural language processing," in [*Proceedings of the 29th international conference on Software Engineering*], 499–510, IEEE Computer Society (2007).

[15] Wen, M., Wu, R., and Cheung, S.-C., "Locus: Locating bugs from software changes," in [*Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*], 262–273, IEEE (2016).

[16] Lam, A. N., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N., "Bug localization with combination of deep learning and information retrieval," in [*Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*], 218–229, IEEE (2017).

[17] Saha, R. K., Lease, M., Khurshid, S., and Perry, D. E., "Improving bug localization using structured information retrieval," in [*Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*], 345–355, IEEE (2013).

[18] Kochhar, P. S., Thung, F., and Lo, D., "Automatic fine-grained issue report reclassification," in [*Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*], 126–135, IEEE (2014).

[19] Murphy-Hill, E., Zimmermann, T., Bird, C., and Nagappan, N., "The design of bug fixes," in [*Proceedings of the 2013 International Conference on Software Engineering*], 332–341, IEEE Press (2013).

[20] Melville, P. and Sindhwani, V., "Recommender systems," in [*Encyclopedia of machine learning*], 829–838, Springer (2011).

[21] Rahman, F. and Devanbu, P., "How, and why, process metrics are better," in [*Software Engineering (ICSE), 2013 35th International Conference on*], 432–441, IEEE (2013).

[22] Liu, T.-Y. et al., "Learning to rank for information retrieval," *Foundations and Trends® in Information Retrieval* **3**(3), 225–331 (2009).

[23] Joachims, T., "Training linear svms in linear time," in [*Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*], 217–226, ACM (2006).

[24] Bordes, A., Bottou, L., Gallinari, P., and Weston, J., "Solving multiclass support vector machines with larank," in [*Proceedings of the 24th international conference on Machine learning*], 89–96, ACM (2007).

[25] Tao, D., Tang, X., Li, X., and Wu, X., "Asymmetric bagging and random subspace for support vector machines-based relevance feedback in image retrieval," *IEEE transactions on pattern analysis and machine intelligence* **28**(7), 1088–1099 (2006).