Rank SVM Based Tracking and Mapping Bug Reports to Relevant Files

Dr. S. Preetha[1], **N. Gangarajam**[2]

[1]Assistant Professor, Department of Computer Science, Sri Ramakrishna College of Arts & Science for Women, Coimbatore District

[2]Research Scholar, Department of Computer Science, Sri Ramakrishna College of Arts & Science for Women, Coimbatore District

Abstract: ***Once the bug occurred, it is a difficult process to localize the bug. It is taking the long time for placing the bug. So the tedious process of placing the bugs taking more time. Sometimes this time taken is more than fixing the bugs.A tool for ranking all the source files of a project with respect to how likely to contain the cause of the bug world enable developers to narrow down their search and potentially could lead to a substantial increase in productivity. Adaptive Rank SVM approach that leverages domain knowledge through functional decompositions of source code files into methods, API descriptions of library components used in the code, the bug-fixing history, and the code changes history. Given a bug report, the ranking score of each source file is computed as a weighted combination of an array of features encoding domain knowledge, where the weights are trained automatically on previously solved bug reports using Learning-to-rank Technique.***

Keywords: Ranking Model, Filtering, Pairwise approach

1. Introduction

A software bug or defect is a coding mistake that may cause an unintended or unexpected behavior of the software component. Upon discovering an abnormal behavior of the software project, a developer or a user will report it in a document, called a bug report or issue report. A bug report provides information that could help in fixing a bug, with the overall aim of improving the software quality. A large number of bug reports could be opened during the development life-cycle of a software product.

Software errors cost the U.S. industry 60 billion dollars a year according to a study conducted by the National Institute of Standards and Technology . One contributing factor to the high number of errors is the limitation of resources for quality assurance (QA). Such resources are always limited by time, e.g., the deadlines that development teams face, and by cost, e.g., not enough people are available for QA. When managers want to spend resources most effectively, they would typically allocate them on the parts where they expect most defects or at least the most severe ones.

Dynamic bug localization techniques suffer from the drawback that they are based on the availability of two control flows — the passing control flow and the failing control flow. This may not be satisfied in real-world scenarios. The static methods, on the other hand, are usually customized to detect irregularities in a particular programming language following a particular coding convention, which makes them rather restrictive in scope.

If the bug report is constructed as a query and the source code files in the software repository are viewed as a collection of documents, then the problem of finding source files that are relevant for a given bug report can be modeled as a standard task in information retrieval (IR). As propose to approach, it as a ranking problem, in which the source files (documents) are ranked with respect to their relevance to a given bug report (query). In this context, relevance is equated with the likelihood that a particular source file contains the cause of the bug described in the bug report.

The ranking function is defined as a weighted combination of features, where the features draw heavily on knowledge specific to the software engineering domain in order to measure relevant relationships between the bug report and the source code file. While a bug report may share textual tokens with its relevant source files, in general there is a significant inherent mismatch between the natural language employed in the bug report and the programming language used in the code.

Ranking methods that are based on simple lexical matching scores have sub optimal performance, in part due to lexical mismatches between natural language statements in bug reports and technical terms in software systems. Our system contains features that bridge the corresponding lexical gap by using project specific API documentation to connect natural language terms in the bug report with programming language constructs in the code.

1.1 Learning to rank

Bug reporting is using the Learning to Rank application which is called as Machine Learned Ranking(MLR). It is the application of Machine learning which is used in Ranking models for Information RetrivalSystem(IR).

The Ranking Model purpose is to rank that is produce a permutation of items in new, unknown list in a way which is “Similar” to ranking in the training data in some sense. Learning to Ranking Algorithms mainly used for IR System but also in some other area also.

Learning to rank has emerged as an active and growing area of research both in information retrieval (IR) and machine learning (ML). The goal of learning to rank is to automatically learn a ranking model from training data, such that the model can sort objects (e.g., documents) according to their degrees of relevance, preference, or importance as defined in a specific application. Many IR problems are by nature ranking problems, and many IR technologies can be potentially enhanced by using learning to rank techniques.

1.2 Application

Ranking is a central part of many information retrieval problems, such as document retrieval, collaborative filtering, sentiment analysis, and online advertising. A possible architecture of a machine-learned search engine is shown in the figure 1 to the right.

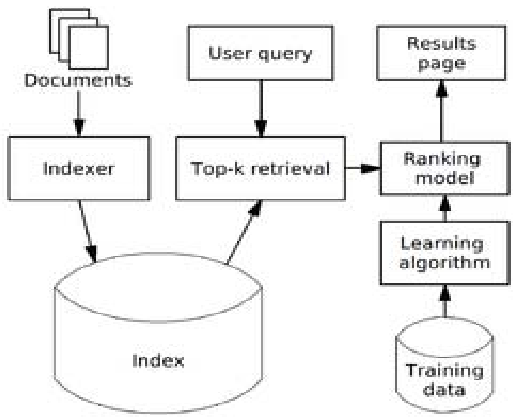


Figure 1: Machine-Learned Ranking Search Engine

Training data is used by a learning algorithm to produce a ranking model which computes the relevance of documents for actual queries. Usually users expect a search query to complete in a very few minutes, which makes it impossible to complex ranking model on each document in the corpus, So two-Phase scheme is used.

First phase a small number of potentially relevant documents are identified using simpler retrieval models which is used for fast query evaluation such as Vector Space Model, Boolean Model weighted AND and BM25.This phase is called as Top-document Retrieval method.

In Second Phase, a more accurate but computationally expensive Machine-Learned model is used to re-rank these documents.

Information Retrieval Quality
Information retrieval quality is usually evaluated by the following three measurements:
1) Precision
2) Recall
3) Average Precision
For a specific query to a database, let be the set of relevant information elements in the database and be the set of the retrieved information elements.

1.3 Approaches in Learning to Rank

A. Point wise approach
In this case it is assumed that each query-document pair in the training data has a numerical or ordinal score. Then learning-to-rank problem can be approximated by a regression problem given a single query-document pair, predict its score. A number of existing supervised machine learning algorithms can be readily used for this purpose. Ordinal regression and classification algorithms can also be used in point wise approach when they are used to predict score of a single query-document pair, and it takes a small, finite number of values.

B. Pair wise approach
In this case learning-to-rank problem is approximated by a classification problem learning a binary classifier that can tell which document is better in a given pair of documents. The goal is to minimize average number of inversions in ranking. Refer to the above technique as pair wise preference ranking or round robin ranking. It is a straight-forward generalization of pair wise or one-against-one classification, aka round robin learning, which solves multi-class problems by learning a separate theory for each pair of classes.

In previous work, Furnkranz (2002) showed that, for rule learning algorithms, this technique is preferable to the more commonly used one-against-all classification method, which learns one theory for each class, using the examples of this class as positive examples and all others as negative examples. Round robin has also been successfully used in other fields, in particular in the area of Support Vector Machines(SVM) (Hsu and Lin, 2002, and references therein. Furnkranz (2002) for a brief survey of related work on pair wise classification.

More importantly, however, Furnkranz (2002) showed that, despite its complexity being quadratic in the number of classes, the algorithm is no slower than the conventional one-against-all technique.

C. The pair wise transform
As proved in (Herbrich 1999), if consider linear ranking functions, the ranking problem can be transformed into a two-class classification problem. For this form the difference of all comparable elements such that our data is transformed into $(x'k,y'k)=(xi-xj,sign(yi-yj))$ for all comparable pairs. This way transformed our ranking problem into a two-class classification problem.

The following plot shows this transformed dataset, and color reflects the difference in labels, and our task is to separate positive samples from negative ones. The hyperplane {x^T w = 0} separates these two classes.

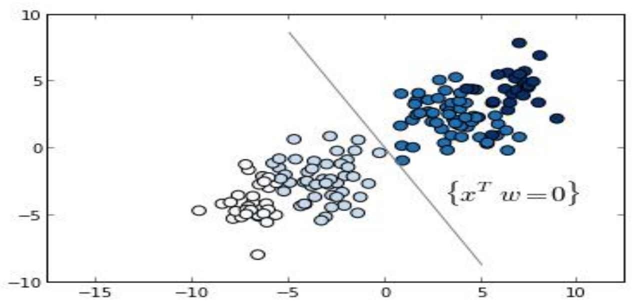


Figure 2: Transformed Dataset

As see in the above figure 2, this classification is separable. This will not always be the case, however, in our training set there are no order inversions, thus the respective classification problem is separable and it will now finally train a Support Vector Machine (SVM) model on the transformed data.

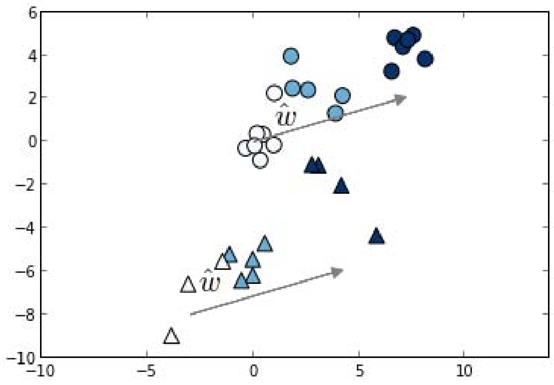


Figure 3: Trained With SvmOn Transformed Dataset

This model is known as RankSVM. will then plot the training data together with the estimated coefficient *w*^ by RankSVM. This Trained SVM Transformed Dataset shown in the below figure 3.

1.4 Ranking SVM

A Ranking SVM is a variant of the support vector machine algorithm, which is used to solve certain ranking problems (via learning to rank). The Ranking SVM algorithm is a learning retrieval function that employs pair-wise ranking methods to adaptively sort results based on how 'relevant' they are for a specific query. The Ranking SVM function uses a mapping function to describe the match between a search query and the features of each of the possible results. This mapping function projects each data pair (such as a search query and clicked web-page, for example) onto a feature space. These features are combined with the corresponding click-through data (which can act as a proxy for how relevant a page is for a specific query) and can then be used as the training data for the Ranking SVM algorithm. Generally, Ranking SVM includes three steps in the training period:

1) It maps the similarities between queries and the clicked pages onto a certain feature space.
2) It calculates the distances between any two of the vectors obtained in step 1.
3) It forms an optimization problem which is similar to a standard SVM classification and solves this problem with the regular SVM solver.

1.5 Building the Benchmarks

This is the dataset that was used to evaluate the learning-to-rank approach. The dataset contains bug reports and the corresponding commit history for six open source Java projects: AspectJ, Birt, Eclipse Platform UI, JDT, SWT, and Tomcat.

For each of the subject systems, as created a benchmark to evaluate the impact analysis techniques. The benchmark consists of a set of change requests that has the following information for each change request: a natural language query (change request summary) and a gold set of methods that were modified to address the change request.

The benchmark was established by a human investigation of the change requests (done by one of the authors), source code, and their historical changes recorded in version-control repositories. Subversion (SVN) repository commit logs were used to aid this process. For example, keywords such as Bug Id in the commit messages/logs were used as starting points to examine if the commits were in fact associated with the change request in the issue tracking system that was indicated with these keywords.

2. Literature Survey

A. Looking For Bugs in All the Right Places

The continue investigating the use of a negative binomial regression model to predict which files in a large industrial software system are most likely to contain many faults in the next release. A new empirical study is described whose subject is an automated voice response system. Not only is this system's functionality substantially different from that of the earlier systems studied (an inventory system and a service provisioning system), it also uses a significantly different software development process. Instead of having regularly scheduled releases as both of the earlier systems did, this system has what are referred to as "Continuous Releases".

These predictions were based on file characteristics that could be objectively assessed, including the size of the file in terms of the number of lines of code (LOC), whether this was the first release in which the file appeared, whether files that occurred in earlier releases had been changed or remained unchanged from the previous release, how many previous releases the file occurred in, how many faults were detected in the file during the previous release, and the programming language in which the file was written.

The model is constructed using logistic regression, and does not predict which parts of the code are most likely to contain faults. Instead it predicts either failure or non failure for a given maintenance request. The model, implemented as a web-based tool available to project management, is used to help schedule the implementation of a given maintenance request, and to determine the level of testing resources to apply to validate the implementation.

The paper presents the regression formulas used to create the prediction model and describes the use of the tool in a software maintenance environment, but does not report a success rate for the model's predictions Graves et al. (2014) performed a study to determine characteristics of modules that are associated with faults, and constructed and evaluated several models for predicting the number of faults that would appear in a future version of the modules. Their study used the fault history of a large telecommunications system containing approximately 1.5 million LOC, organized into 80 modules, containing a total of about 2500 files. The prediction models were used to make fault predictions for a single two year time interval, based on the system's history for the preceding two years. They found that module size was a poor predictor of fault likelihood, while the most accurate predictors included combinations of the module's age, the number of changes made, and the ages of the changes. The authors also described the application of their models to a one year interval in the middle of the original two year interval and found that certain parameter values differed by an order of magnitude between the two time periods. Our results in and in the present paper partly agree and partly conflict with. File age and previous changes were positive indicators of fault-proneness in our studies as well as that of, but in contrast to, have consistently found file size to be a strong predictor of fault-proneness.

3. Methodology And Implementation

A. Existing System

In learning to rank a number of categories are given and a total order is assumed to exist over the categories. Labeled instances are provided. Each instance is represented by a feature vector, and each label denotes a rank. Existing methods fall into two categories. They are referred to in this paper as "point-wise training" and "pair-wise training". In point-wise training, each instance (and its rank) is used as an independent training example. The goal of learning is to correctly map instances into intervals.

A tool for ranking all the source files of a project with respect to how likely they are to contain the cause of the bug world enable developers to narrow down their search and potentially could lead to a substantial increase in productivity. Adaptive ranking approach that leverages domain knowledge through functional decompositions of source code files into methods, API descriptions of library components used in the code, the bug fixing history, and the code change history.

Given a bug report, the ranking score of each source file is computed as a weighted combination of an array of features encoding domain knowledge, where the weights are trained automatically on previously solved bug reports using a learning –to-rank technique.

B. Vector Space Representation

If suppose regard the bug report as a query and the source code file as a text document, then can employ the classic Vector Space Model(VSM) for ranking, a standard model used in information retrieval. In this model, both the query and the document are represented as vectors of term weights.

Given an arbitrary document d(a bug report or a source code file), compute the term weights wt,d for each term t in the vocabulary based on the classical tf.idf weighting scheme in which the term frequency factors are normalized.

The term frequency factor tf (t,d) represents the number of occurrences of term t in document d, whereas the document frequency factor dft represents the number of documents in the repository that contains term t. In VSM, a bug report use both its summary and description to create the VSM representation. For a source file, use its whole content-code and comments. To tokenize an input document, first split the text into a bag of words using white space. Then remove punctuation, numbers and standard IR stop words such as conjunctions or determiners. In general, most of the text in a bug report is expressed in natural language (eg. English), whereas most of the content of a source code file is expressed in a programming language (eg.java). Since the inner product used in the cosine similarity function has non-zero terms only for tokens that are in common between the bug report and the source file, this implies that the surface lexical similarity feature described 1) the source code has expensive, comprehensive comments or 2) the bug report includes snippets of code or programming language constructs such as names of classes or methods.

For each method in a source file, extract a set of class and interface names from the explicit type declarations of all local variables. Using the project API specification, the textual descriptions of these classes and interfaces, including the descriptions of all their direct or indirect super classes or super interfaces.

For each method m create a document m.api by concatenating the corresponding API descriptions. Finally, take the API specifications of all methods in the source file s and concatenate them into an overall document.

While a bug report may share textual tokens with its relevant source files, in general there is a significant inherent mismatch between the natural language employed in the bug report and the programming language used in the code.

Ranking methods that are based on simple lexical matching scores have suboptimal performance, in part due to lexical mismatches between natural and programming language statements in bug reports and technical terms in software systems.

The resulting ranking function is a linear combination of features, whose weights are automatically trained on previously solved a bug reports using a learning-to-ranking techniques.

To avoid contaminating the training data with future bug-fixing information from previous reports, created fine-grained benchmarks by checking out the before –fix versions of the project for every bug report.

Drawbacks of Vector space Model

1) Weighting is intuitive but not very formal.
2) The order in which the terms appear in the document is lost in the vector space representation.

C. Proposed System

Ranking SVM is a typical method of learning to rank. Then point out that there are two factors one must consider when applying Ranking SVM, in general a "learning to rank" method, to bug mapping. First, correctly ranking bugs on the top of the result list is crucial for an Information Retrieval system. One must conduct training in a way that such ranked results are accurate. Second, the number of relevant bugs can vary from query to query. One must avoid training a model biased toward queries with a large number of relevant bugs.

D. Metrics: Support Vector Machine (SVM) Representation

Support Vector Machine (SVM) is a classification technique based on statistical learning theory. A support vector machine constructs a hyper plane or set of hyper planes in a high or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyper plane that has the largest distance to the nearest training data point of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

SVM is a machine-learning method, based on the principle of structural risk minimization, which performs well when applied to data outside the training set. They formulate MC detection as a supervised-learning problem and apply SVM to develop the detection algorithm.

The proposed algorithm works as follows:
1) Create a set of bug reports one from each repository in such a way that the similarity score between any pair or reports is smaller than the specified threshold. Label this repository as the set of negative examples.

2) Create a set of bug reports in such a way that the similarity score between each one of them is greater than the specified threshold. This can be formed using a single master reports and all its duplicates. However, bug reports from other repositories are to be prioritized having nearly equal score as that of the master and duplicates.

3) Name this set as training set and train an SVM based classifier on this set.

4) Test the accuracy of SVM on any incoming bug report. Positive and negative examples are created using this bucket structure for training of the support vector machine, which is a linear classifier. Positive examples can be created using a master bug report and its duplicates, or two duplicates from the same bucket. Negative examples can be created using reports from distinct buckets. Thus the number of negative examples fairly exceeds the number of positive examples. Therefore, the negative examples should be chosen suitably to accommodate nearly all the distinct pairs.

In above Point 4, results corresponding to Java bug repository are considered. Topic modeling is done using SVM to model the topics which are non-functional requirements of the software. Textual and categorical features are analyzed along with the semantic features to extend the feature set and to perform triaging more accurately.

If regard the bug report as a query and the source code file as a text document, then can employ the Support Vector Machine (SVM) for ranking, a standard model used in information retrieval. In this model, both the query and the document are represented as vectors of term weights.

Given an arbitrary document d (a bug report or a source code file), compute the term weights w t,d for each term t in the vocabulary based on the classical tf.idf weighting scheme in which the term frequency factors are normalized, as follows:

(i) Surface Lexical Similarity

For a bug report, use both its summary and description to create the SVM representation. For a source file, use its whole content – code and comments. To tokenize an input document, first split the text into a bag of words using white spaces. Then remove punctuation, numbers, and standard IR stop words such as conjunctions or determiners.

Compound words such as “WorkBench” are split into their components based on capital letters, although more sophisticated methods such as could have been used here too. The bag of words representation of the document is then augmented with the resulting tokens – “Work” and “Bench” in this example – while also keeping the original word as a token. Finally, all words are reduced to their stem using the Porter stemmer, as implemented in the NLTK 1 package.

This process will reduce derivationally related words such as “programming” and “programs” to the same stem “program”, which is known to have a positive impact on the recall performance of the final system.

E. Class Name Similarity

A bug report may directly mention a class name in the summary, which provides a useful signal that the corresponding source file implementing that class may be relevant for the bug report. Our hypothesis is that the signal becomes stronger when the class name is longer and thus more specific

F. Collaborative Filtering Score

It has been observed in that a file that has been fixed before may be responsible for similar bugs. For example, these three reports describe similar defects and therefore share many keywords with report 378535 Consequently, it is not surprising that source file StackRenderer.java.

G. File Revision History

The source code change history provides information that can help predict fault-prone files. For example, a source code file that was fixed very recently is more likely to still contain bugs than a file that was last fixed long time in the past, or never fixed.

H. Structural Information Retrieval

By computing similarities with each method and then maximizing across all methods in a source file, feature φ 1 alleviates the problem of the small similarities that result for localized bugs, when using a straightforward cosine similarity formula in which the normalization factor is correlated with the length of the file.

A related problem may occur when the bug report is very similar with a particular type of content from a source file (e.g. comments, method names, or class names) and dissimilar with everything else, yet the cosine similarity with the entire file is very small due to its large size. To model such cases, follow the structural IR approach of that all., in which a source code file s is parsed into four document fields: all class names in s.class, all methods names in s.method, all variable names in s.variable, and all comments in s.comment.

For example, a field such as s.method is equivalent with a document that contains all the method names defined in the source code file s. The summary and the description of a bug report r are used to create two query fields: r.summary and r.description, respectively.

I. The File Dependency Graph

Expect complex code to be more prone to bugs than simple code. Thus, the complexity of the source code contained in a file can provide another useful signal with respect to the likelihood that the file contains bugs. An accurate measure of code complexity would require a good representation of the semantics of the code.
Since a comprehensive semantic analysis of code is currently not feasible, resort to a characterization of code complexity based on syntactic features. For example, a proxy measure for the complexity of a source code file can be defined.

J. Feature Scaling

Features with widely different ranges of values are detrimental in machine learning models. Feature scaling helps bring all features to the same scale so that they become comparable with each other.

K. Implementation

The implementation is performed for the given technique using Java as the programming language and MySql as the local database to store the processed information. The implementation is performed in the five stages.

1) Retrieving and Parsing the Software Bugs: In the first step the software bugs are retrieved at local system and parsing using tokenization for extracting the bug attributes and their corresponding values.
2) Creating local database for selected attributes: The extracted attributes are filtered for analysis and saved in the local database.
3) Eliminating the possible spams: In this stage the textual bug attributes are analyzed for possible spam. The information which is likely to be spam is ignored and never used for calculating the ranking of team members.
4) Generating Metadata for Ranking: Once the possible spams are eliminated the next step is to prepare the metadata for ranking. This includes counting the number of team members; number of bugs for each member, number of comments for each member etc. is performed in this stage.
5) Implementing the ranking algorithm: With the help of metadata generated in previous stage and using various user implemented in java.

4. Results and Discussion

The first step in our adapted system is to rank all the source code files for every bug report in the dataset. The ranking performance on these 45 Eclipse bug reports is lower than the performance, which was obtained on the 6,495 Eclipse bug reports from our fine-grained benchmark dataset.

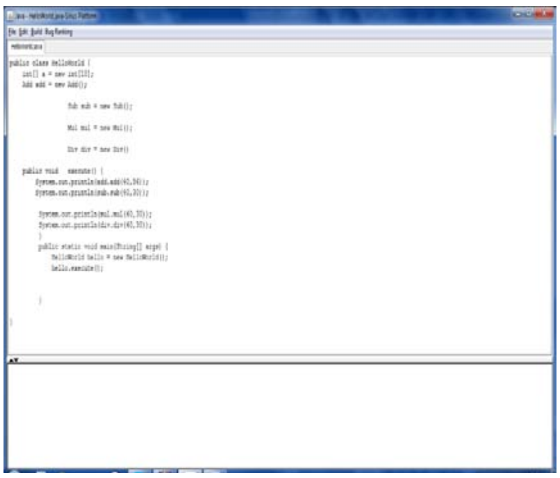
The main reason for this difference is that the 45 bug reports are from 2004 and therefore there is not much historical information that can be used for computing features that are based on collaborative filtering or the file revision history. In particular, there is less opportunity for exploiting duplicated bug reports.

Use a dataset of 157 bugs from 4 popular software projects to evaluate our approach against the baselines. These projects are AspectJ, Ant, Lucene, and Rhino. All four projects are medium-large scale and implemented in Java. AspectJ, Ant, and Lucene contain more than 300 kLOC, while Rhino contains almost 100 kLOC. Table 3 describes detailed information of the four projects in our study.

The 41 AspectJ bugs are from the iBugs dataset which were collected by Dallmeier and Zimmermann . Each bug in the iBugs dataset comes with the code before the x (pre- x version), the code after the x (post- x version), and a set of test cases. The iBugs dataset contains more than 41 AspectJ bugs but not all of them come with failing test cases. Test cases provided in the iBugs dataset are obtained from the various versions of the regression test suite that comes with AspectJ.

The remaining 116 bugs from Ant, Lucene, and Rhino are collected by ourselves following the procedure used by Dallmeier and Zimmermann. For each bug, collected the pre- x version, post- x version, a set of successful test cases, and at least one failing test case.

For example the number of classes and function are calling from the main program then the calling of methods and programs are showing the results as follows.



```
public class HelloWorld {
    int[] a = new int[10];
    Add add = new Add();

            Sub sub = new Sub();

            Mul mul = new Mul();

            Div div = new Div()

    public void   execute() {
        System.out.println(add.add(40,30));
        System.out.println(sub.sub(40,30));

         System.out.println(mul.mul(40,30));
         System.out.println(div.div(40,30));
         }
         public static void main(String[] args) {
             HelloWorld hello = new HelloWorld();
             hello.execute();


         }

}
```

Figure 4: Calling Main Program Example

Here some of the sample programs are calling from the main program. From the main program what are all the sub programs are calling and what are all the sub programs are interlinked with the Main program all the details are showing in the following Parser Dependency Graph figure 5.

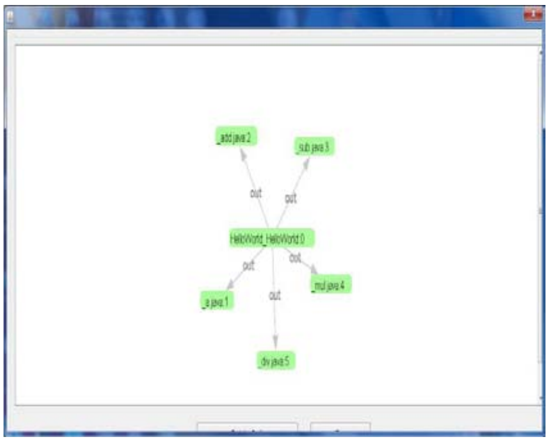
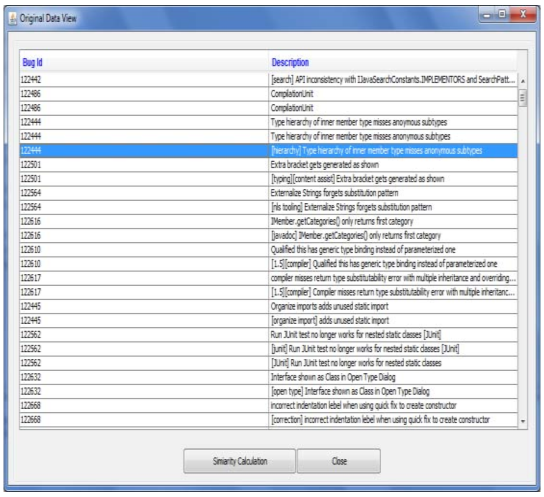


Figure 5: Parser Dependency Graph

From this figure 5, the dependency of the sub programs are visualized. Once the program ready to execute then, the Benchmark Dataset loaded for the comparision of the Bug similarity Calculation. This will lead to the program to compare with the expected bugs which is available in the Original Data View as shown in the figure 6. So that Bug Id and Description will shown clearly in the output for the resultant window. After the similarity calculation the time comparion will happen.



Bug Id	Description
122442	[search] API inconsistency with IJavaSearchConstants.IMPLEMENTORS and SearchPatt...
122486	CompilationUnit
122486	CompilationUnit
122444	Type hierarchy of inner member type misses anoymous subtypes
122444	Type hierarchy of inner member type misses anonymous subtypes
122444	[hierarchy] Type hierarchy of inner member type misses anonymous subtypes
122501	Extra bracket gets generated as shown
122501	[typing][content assist] Extra bracket gets generated as shown
122564	Externalize Strings forgets substitution pattern
122564	[nls tooling] Externalize Strings forgets substitution pattern
122616	IMember.getCategories() only returns first category
122616	[javadoc] IMember.getCategories() only returns first category
122610	Qualified this has generic type binding instead of parameterized one
122610	[1.5][compiler] Qualified this has generic type binding instead of parameterized one
122617	compiler misses return type substitutability error with multiple inheritance and overriding...
122617	[1.5][compiler] Compiler misses return type substitutability error with multiple inheritanc...
122445	Organize imports adds unused static import
122445	[organize import] adds unused static import
122562	Run JUnit test no longer works for nested static classes [JUnit]
122562	[junit] Run JUnit test no longer works for nested static classes [JUnit]
122562	[JUnit] Run JUnit test no longer works for nested static classes
122632	Interface shown as Class in Open Type Dialog
122632	[open type] Interface shown as Class in Open Type Dialog
122668	incorrect indentation lebel when using quick fix to create constructor
122668	[correction] incorrect indentation lebel when using quick fix to create constructor

Figure 6: Original Data View

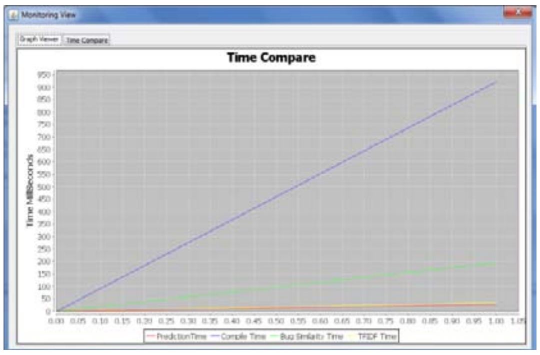
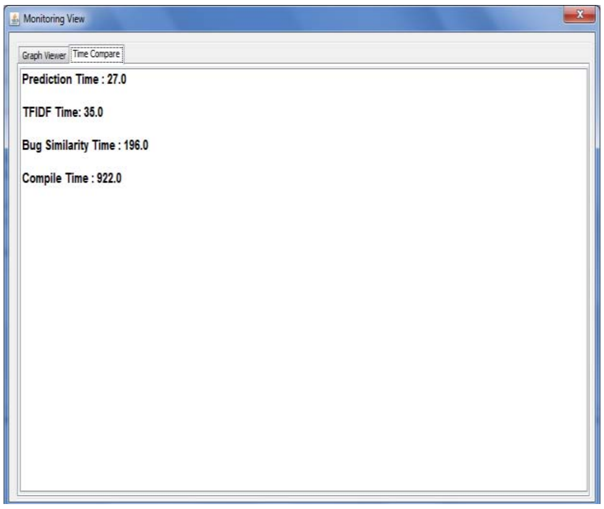


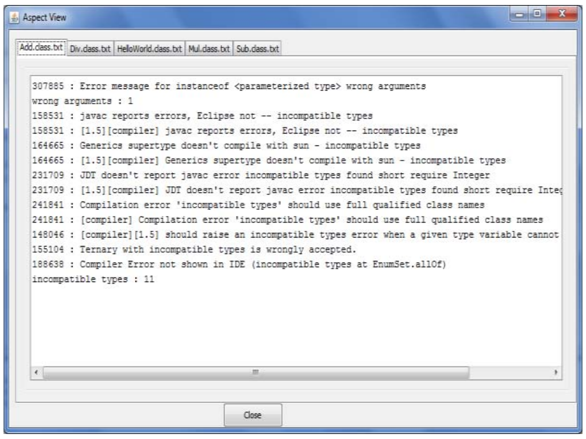
Figure 7: Time Comparision –Graph view



```
Prediction Time : 27.0

TFIDF Time: 35.0

Bug Similarity Time : 196.0

Compile Time : 922.0
```

Figure 8: Time Comparision-Result View

From the figure 7 shows the Time Comparision Graph View. From this figure 8, what is the Prediction Time and Bug Similarity Time,Compile Time and The Term Frequency vs Document Frequency Time all the details are showing in milliseconds.



```
307885 : Error message for instanceof <parameterized type> wrong arguments
wrong arguments : 1
158531 : javac reports errors, Eclipse not -- incompatible types
158531 : [1.5][compiler] javac reports errors, Eclipse not -- incompatible types
164665 : Generics supertype doesn't compile with sun - incompatible types
164665 : [1.5][compiler] Generics supertype doesn't compile with sun - incompatible types
231709 : JDT doesn't report javac error incompatible types found short require Integer
231709 : [1.5][compiler] JDT doesn't report javac error incompatible types found short require Inteç
241841 : Compilation error 'incompatible types' should use full qualified class names
241841 : [compiler] Compilation error 'incompatible types' should use full qualified class names
148046 : [compiler][1.5] should raise an incompatible types error when a given type variable cannot
155104 : Ternary with incompatible types is wrongly accepted.
188638 : Compiler Error not shown in IDE (incompatible types at EnumSet.allOf)
incompatible types : 11
```

Figure 9: Output Image

The above figure 9 shows the expected Bug similarity Result for the Addition functions and this will happen for all the sub programs what are all the programs are interlinked with the main programs and the same process is happening for all the classes and methods.

A failing test case is often included as an attachment to a bug report or committed along with the x in the post- x version. When a developer receives a bug report, he/she needs to replicate the error described in the report. In this process, he is creating a failing test case. Unfortunately, not all test cases are documented and saved in the version control systems.

5. Conclusion and Future Enhancement

Locating bugs is important, difficult, and expensive, particularly for large-scale software projects. To address this, natural language information retrieval (IR) techniques are increasingly being used to suggest potential faulty source files given bug reports. While these techniques are very scalable, in practice their effectiveness remains low in accurately localizing bugs to a small number of files.

Our key insight is that structured IR-based on code constructs, such as class and method names, enables more accurate bug localization. Our Project embodies this insight, builds on an open source IR toolkit, requires only the source code and bug reports, and takes advantage of bug similarity data if available. When bug similarity data is not used, the off-the-shelf IR took it (unmodified) already exceeds state-of-the-art tool, Bug Locator's accuracy.

In our future research, would like to explore the following areas to further improve our model: bug report summarization and learning parameters. Bug Report Summarization. In this paper, showed how the performance of bug localization improves by focusing on condensed information such as bug summaries, class names, or method names.

However, still used exactly the same long bug descriptions from bug reports. Such summarized bug descriptions may further improve the performance of bug localization.

References

[1] G. Antoniol and Y.-G.Gueheneuc, "Feature identification: A novel approach and a case study," in Proc. 21st IEEE Int. Conf. Softw.Maintenance,Washington, DC, USA, 2005, pp. 357–366.
[2] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories,"in Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng., NewYork, NY, USA, 2010 pp. 157–166.
[3] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for bugs in all the right places," in Proc. Int. Symp. Softw.Testing Anal., NewYork, NY, USA, 2006, pp. 61–72.
[4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in Proc. 15th Int.Conf. Softw. Eng., Los Alamitos, CA, USA, 1993, pp. 482–498.
[5] D. Binkley and D. Lawrie, "Learning to rank improves IR in SE,"in Proc. IEEE Int. Conf. Softw. Maintenance Evol., Washington, DC,USA, 2014, pp. 441–445.
[6] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis,"in Proc. 6th IEEE Int. Working Conf. Mining Softw. Repositories,Washington, DC, USA, 2009, pp. 71–80.
[7] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in Proc. 11th Working Conf. Reverse Eng., Washington, DC, USA, 2004,pp. 214–223.
[8] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," Empirical Softw. Eng., vol.18, no. 2,pp. 277–309, 2013.
[9] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," J. Softw.: Evol.Process, vol. 25, no. 1, pp. 53–95, 2013.
[10] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk,"Enhancing software traceability by automatically expanding corpora with relevant documentation," in Proc.IEEE Int. Conf. Softw. Maintenance, Washington, DC, USA,2013, pp. 320–329.
[11] H. Cleve and A. Zeller, "Locating causes of program failures," in Proc. 27th Int. Conf. Softw. Eng., New York, NY, USA, 2005,pp. 342–351.