**BUG REPORT CLASSIFICATION**

Apurva Aggarwal 16103031

SomilRastogi 16103193

Ajay Kumar Kushwaha 16104013

**Supervisor :** Mrs.Sangeeta Lal



**Dec-2019**

**Submitted in partial fulfilment of the Degree of**

**Bachelor of Technology**

**in**

**Information Technology**

**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING & INFORMATION TECHNOLOGY**

**JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA**

**(I)**

**TABLE OF CONTENTS**

**CHAPTER TOPICS PAGE NO.**

**NO.**

**Student Declaration 4**

**Certificate from the supervisor 5**

**Acknowledgement 6**

**Summary 7**

**List of Figures 8**

**List of Tables 10**

**Chapter-1 Introduction**

1.1 General Introduction 11

1.2 Problem Statement 11

1.3 Significance/Novelty of the problem 11

1.4 Empirical Study 12

1.5 Brief Description of the Solution Approach 15

1.6 Comparison of existing approaches 15

to the problem framed

**Chapter-2** **Literature Survey**

2.1 Summary of papers studied 17

2.2 Integrated summary of the literature studied 33

**Chapter 3: Requirement Analysis and Solution Approach**

3.1 Overall description of the project 34

3.2 Requirement Analysis 35

3.3 Solution Approach 36

**Chapter-4 Modelling and Implementation Details**

4.1 Design Diagrams

4.1.1Use Case diagrams 38

4.1.2 Class diagrams / Control Flow Diagrams 39

4.1.3 Sequence Diagram/Activity diagrams 41

4.2 Implementation details and issues 42

4.3 Risk Analysis and Mitigation 47

**Chapter-5 Testing (Focus on Quality of Robustness and Testing)**

5.1 Testing Plan 49

5.2 Component decomposition and 50

type of testing require

5.3 List all test cases in prescribed format 50

5.5 Limitations of the solution 51

**Chapter-6 Findings, Conclusion, and Future Work**

6.1 Findings 52

6.2 Conclusion 58

6.3 Future Work 59

**References 60**

**(II)**

**DECLARATION**

We hereby declare that this submission is our own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Place: Jaypee Institute of Information Technology, Sec-62, Noida

Date: 13th December, 2019

Name(s): Apurva Aggarwal SomilRastogi Ajay Kumar Kushwaha

Enrolment No(s): 1610303116103193 16104013

Signature(s):\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

**(III)**

**CERTIFICATE**

This is to certify that the work titled “**Bug Report Classification**” submitted by “**Apurva Aggarwal, Somil Rastogi & Ajay Kumar Kushwaha**” in partial fulfilment for the award of degree of B.Tech. from Jaypee Institute of Information Technology, Noida, has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Name of Supervisor: Mrs. Sangeeta Lal

Designation: Assistant Professor (Senior Grade)

Date: 13th December 2019

**(IV)**

**ACKNOWLEDGEMENT**

We take this honourable chance to express our significant feeling of appreciation and gratefulness to each and every individual who helped us throughout the term on this venture. We want to express our uncommon thanks of appreciation to our mentor Mrs. Sangeeta Lal, who gave us the brilliant chance to be a piece of this magnificent venture and for giving direction and master supervision to this undertaking. She has set a tremendous case on us youthful receptive personalities in the span of making this venture. We are truly grateful to her.

Date: 13th December, 2019

Name(s): Apurva Aggarwal SomilRastogi Ajay Kumar Kushwaha

Enrolment No(s): 16103031 16103193 16104013

Signature(s): \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

**(V)**

**SUMMARY**

The project titled **Bug Report Classification** has been accomplished as our major project. Among numerous neural network architectures, particularly interesting architecture was introduced by Finish Professor TeuvoKohonen in the 1980s, **self-organizing map (SOM),** sometimes also called a Kohonen maps. As a particular type of artificial neural networks, self-organizing maps (SOMs) are trained using an unsupervised, competitive learning to produce a low dimensional, discretized representation of the input space of the training samples. Self-organizing maps are known for its clustering, visualization and classification capabilities.We have applied SOM on Chicago Crime Data Set to identify crime hotspots in the area.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Signature of Student(s) Signature of Supervisor

**(VI)**

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **FIGURE NO.** | **TITLE** | **PAGE NO.** |
| 1 | Class Distribution | 13 |
| 2 | TFIDF Score Chart | 14 |
| 3 | Top 10 words in each Bug Issue | 15 |
| 4 | MTTR of Issue Type | 15 |
| 5 | Use Case Diagram | 39 |
| 6 | Control Flow Diagram | 42 |
| 7 | Activity Diagram | 43 |
| 8 | Simplified Random Forest | 46 |
| 9 | Ensemble Learning | 48 |
| 10 | Bernoulli Naïve Bayes | 54 |
| 11 | Multinomial Naïve Bayes | 54 |
| 12 | Random Forest | 55 |
| 13 | SVM | 55 |
| 14 | Comparison of Training time | 56 |
| 15 | Comparison of Testing Time | 56 |

|  |  |  |
| --- | --- | --- |
| 16 | Comparison of Accuracy | 57 |
| 17 | Ensemble Learning (Hard) | 57 |
| 18 | Ensemble Learning (Soft) | 58 |
| 19 | Bagging | 58 |
| 20 | Adaboost | 59 |
| 21 | Stochastic gradient boosting | 59 |
| 22 | Overall Comparison | 60 |

**(VII)**

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| **S.NO.** | **TITLE** | **PAGE NO** |
| **1** | Risk Analysis and Mitigation | 49 |
| **2** | Testing Plan | 51 |
| **3** | Component Decomposition and type of testing required | 52 |
| **4** | List of all Test Cases | 52 |

**CHAPTER-1**

**INTRODUCTION**

Issue Tracking System (IST) such as JIRA and Bugzilla are used widely these days to collect the reports of issues that occur during development and maintenance phase of the software. Once bugs are collected developers assigned them various labels like priority, type, component etc., and triage them to appropriate developers for its resolution. However, this manual resolution has two main challenges. First, a large number of issues are reported in the system daily. Nearly 300 issues are reported daily is the system. Assigning various labels to such a large of issues is a tedious and time consuming task. Second, Manual assignment often leads error in labelling.  40% all the issues are assigned a wrong type to the bug report. This wrong assignment of labels often leads to increase in fixing time. Hence, an automated system that can assign labels to bug report will be great use. A deeper understanding of the properties and features of various categories of bug reports can have implications in improving software maintenance processes, tools and practices. Because of wrong categorization of bug reports, often valuable time of software developers gets wasted. It will be beneficial for the software developers, if a system can automatically classify a bug report.

**Problem Statement**

Issue reports are submitted on the JIRA issue tracking system. They contain various fields such as report id, report summary, report description, project description, comments, date of creation, date of modification, date of resolution, issue type etc.  The bug reports fall into 7 categories of Issue Type namely- bug, Improvement, Feature Request, Task, Sub-Task, Test and Wish. The main aim of our research is to perform an empirical analysis (by identifying several metrics and characteristics serving as dimensions on which various types of reports can be compared) and build a model that classifies the reports automatically into these 7 categories.

**Novelty of the problem**

Modern version control systems include bug tracking mechanisms that developers can use to highlight the presence of bugs. Software developers, testers and customers routinely submit issue reports to software issue trackers to record the problems they face in using a software. The issues are then directed to appropriate experts for analysis and fixing. This is done by means of bug reports, i.e., textual descriptions reporting the problem and the steps that led to a failure. In past and recent years, the research community deeply investigated methods for easing bug triage, that is, the process of assigning the fixing of a reported bug to the most qualified developer. Nevertheless, only a few studies have reported on how to support developers in the process of understanding the type of a reported bug, which is the first and most time-consuming step to perform before assigning a bug-fix operation.

**Empirical study**

**Questions Answered Through Empirical Study**

1. What is the distribution of different categories (for example, bug, documentation, improvement etc.) among all the issue reports?
2. Are there any distinguishing terms that differentiate various issue categories?
3. Is there any significant difference between Mean Time to Repair (MTTR) of different issue categories?

We identify the percentage of each bug report type. The overall distribution of different bug report type will provide important insights about how frequently each type of bug report occurs in the system.

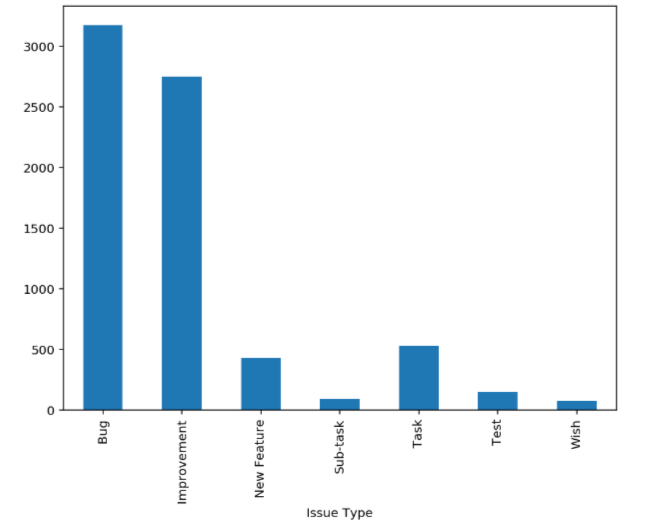


Fig-1

We analyze the textual content of issue reports belonging to different categories. Our aim is to identify to whether different issue categories have differentiating terms of not. If different issues categories have different terms that textual content of issue reports can be used to build models for automated issue category prediction.

We extract the title and summary of for each issue report. Now, we apply the basic pre-processing techniques to clean the data. We then form the corpus of each issue category. Now, we apply the tf-idf function on each corpus. The tf-idf function is a popular method that is useful in extracting important terms present in a given corpus. Equation1. shows the formula for tf-idf function.

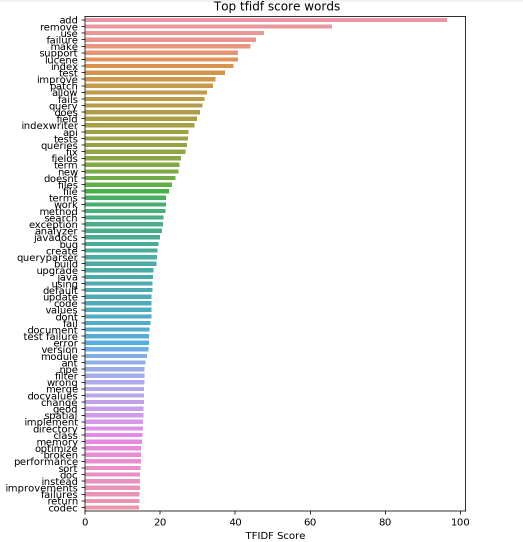


Fig-2

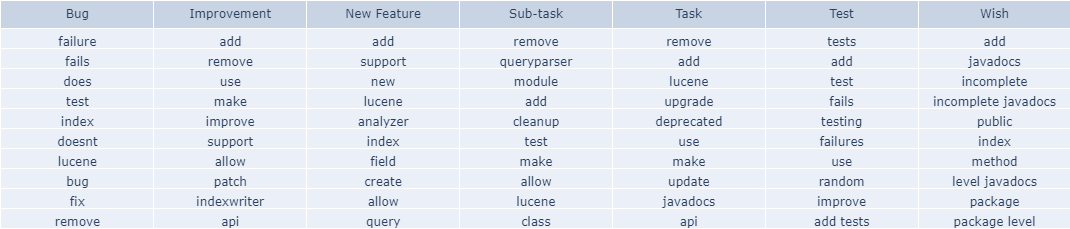


Fig-3

**Top 10 Words in Each Bug Report Issue Type**

Mean Time to Repair (MTTR) indicates how much time it takes to fix a particular issues, i.e., difference between the date when the issues is fixed and the date to which the issue is filed. MTTR can throw light on which issue category requires maximum time to fix. This can be beneficial in prioritizing the issues.

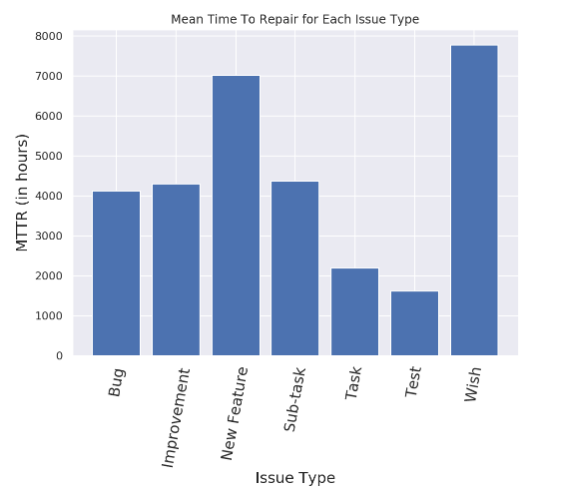


Fig-4

Brief description of the solution approach

* Initially we have got 13 columns which play a significant role in detecting issue type. Here feature selection plays a major role in extracting the relevant features being used. Applying feature selection will help to eliminate features which are less significant.
* Two of the features namely ‘Summary’ and ‘Description’ were found to be the most useful. They were merged and loaded into a single variable.
* We took the corpus to model ‘Bag of Words Model’. We create a dictionary of words and their occurrence calculated for each document in the corpus. Then using term frequency and inverse document frequency formulas,we represented our corpus of words in document vectors.
* Initially for model development, the machine learning algorithms we used and compared were Bernoulli and Multinomial Naive Bayes. As each feature variable has short length, and size of dictionary created is very small, the Naive Bayes algorithm doesn’t perform so well on the dataset.
* Then we used Random Forests and SVM on the same, which provide us better results. All these algorithms are compared on different metrics like accuracy, F1-Score, training and testing time.
* These learning algorithms perform the task of searching through a hypothesis space to find a suitable hypothesis that will make good predictions with aparticular problem. But individually all the learning algorithms do not produce good results.
* So we use Ensemble to combine multiple hypotheses to form a (hopefully) better hypothesis. Ensemble methods we have used –
  + 1. Majority Voting (Hard Voting)
    2. Average Voting (Soft Voting)
    3. Bagging of Decision Trees
    4. AdaBoost
    5. Stochastic Gradient Descent Boosting

Comparison of existing approaches to the problem framed

* Existing work generally focuses on less number of issue report types namely performance, security etc. Two dimensional study concerning such classes have been performed. Also there has been focus on classification of reports into ‘bug’ and ‘not a bug’ class. They analyse the effect of different parameters on the classifiers’ predictive power. But the accuracy achieved with such algorithms individually is quite less. This is also due to the fact that the dataset in itself poses a challenge due to its scarcity.
* Existing approaches for multiclass classification involves usage of various ML algorithms like naive Bayes (NB), k-nearest neighbors (kNN), decision tree (DT) and random forest (RF) separately.
* Bug triage is converted into text classification, they focus on the feature selection algorithms in text data. In this paper, four well-performed algorithms  are chosen in text data and software data, namely Information Gain (IG), x2 statistic (CH), Symmetrical Uncertainty attribute evaluation(SU), and Relief-F Attribute selection (RF).  Based on feature selection, words in bug reports are sorted according to their feature values and a given number of words with large values are selected as representative features.

**CHAPTER-2**

**LITERATURE SURVEY**

**2.1) SUMMARY OF PAPERS STUDIED**

**TABLE 1**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Comparison of Seven Bug Report Types: ACase-study of Google Chrome Browser Project** |
| **AUTHORS** | SangeetaLal and Ashish Sureka |
| **YEAR** | 2012 |
| **INTRODUCTION** | Bug reports submitted to an issue tracking system can belong to different categories such as crash, regression, security, cleanup, polish, performance and usability. A deeper understanding of the properties and features of various categories of bug reports can have implications in improving software maintenance processes, tools and practices. They identify several metrics and characteristics serving as dimensions on which various types of bug reports can be compared. They perform a case-study on Google Chromium Browser open-source project and conduct a series of experiments to calculate various metrics. |
| **METHODS** | They present a characterization study comparing different types of bug reports on metrics such as: statistics on close-time, number of stars, number of comments, discriminatory and frequent words for each class, entropy across reporters, entropy across component, opening and closing trend, continuity and debugging efficiency performance characteristics. The calculated metrics shows the similarities and differences for seven different types of bug reports. Across multiple dimensions within the same project to increase understanding of different bug report types.The study provides evidences of correlation between certain terms and bug report types. Empirical results demonstrating similarities and difference in bug fixing process quality, continuity and efficiency across different bug reports are presented. |
| **CONCLUSION** | Median MTTR of crash and regression bugs is lowest. Performance and usability belongs to high entropy and low incident region. Large number of regression bugs are usability bugs. Terms present in bug report title and description are related to bug type. Description length of crash bug is largest. Large number of crash bugs are resolved as Won’tFix and duplicate. Fixing security bugs involves highest amount of discussion. Milestone change for usability bugs is highest and for cleanup bugs is lowest. Bug fixing process is of high quality among all the bug types (for Google Chrome Project). |

**TABLE 2**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Not All Bugs Are the Same:Understanding, Characterizing, and Classifying Bug Type** |
| **AUTHORS** | Gemma Catolino, Fabio Palomba, Andy Zaidman,andFilomenaFerrucci |
| **YEAR** | 2019 |
| **INTRODUCTION** | Modern version control systems, e.g., GitHub, include bug tracking mechanisms that developers can use to highlight the presence of bugs. This is done by means of bug reports, i.e., textual descriptions reporting the problem and the steps that led to a failure. In past and recent years, the research community deeply investigated methods for easing bug triage, that is, the process of assigning the fixing of a reported bug to the most qualified developer. Nevertheless, only a few studies have reported on how to support developers in the process of understanding the type of a reported bug, which is the first and most time-consuming step to perform before assigning a bug-fix operation. |
| **METHODS** | In this paper, they targetthe problem of understanding the type of a reported bug in two ways: first, they analyze 1,280 bug reports of 119 popular projects belonging to three ecosystemssuch as Mozilla, Apache, and Eclipse, with the aim of building a taxonomy of the types of reported bugs; then, theydevise and evaluate an automated classification model able to classify reported bugs according to the defined taxonomy.They built their classification model by training their classifier using the textual content of the bug report to predict its type. They empirically evaluate the classification model by running it against the dataset coming as output of the taxonomy building phase, measuring its performance adopting a 100 times 10-fold cross validation methodology in terms of F-Measure, AUC-ROC, and Matthew’s Correlation Coefficient (MCC).The results of the study highlight nine different bug reported in bug reports, that span across a broad set of issues (e.g., GUI-related vs Configuration bugs) and are widespread over the considered ecosystems. |
| **CONCLUSION** | They found nine main common bug types over the considered systems. Moreover, thieir model achieves high F-Measure and AUC-ROC (64% and 74% on overall, respectively). |

**TABLE 3**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Automated classification of software issue reports using machine learning techniques: an empirical study** |
| **AUTHORS** | Nitish Pandey, Debarshi Kumar Sanyal, AbirHudait, and Amitava Sen |
| **YEAR** | 2017 |
| **INTRODUCTION** | Software developers, testers and customers routinely submit issue reports to software issue trackers to record the problems they face in using a software. The issues arethen directed to appropriate experts for analysis and fixing. However, submitters often misclassify an improvement request as a bug and vice versa. This costs valuable developer time. Hence automated classification of the submitted reports would be of great practical utility. In this paper, they analyze how machine learning techniques may be used to perform this task. |
| **METHODS** | They build different models to classify issue reports from Jira into bug and non-bug categories in a completely automated way. The issue reports selected belong to three open-source projects HttpClient,Lucene and Jackrabbit. They use the following classification algorithms as implemented in R7: naive Bayes (NB), linear discriminant analysis (LDA), k-nearest neighbours (kNN), support vector machine (SVM) with various kernels, decision tree (DT) and random forest (RF) separately. They analyze the effect of different parameters on the classifiers’ predictive power. They measure classification efficiency in terms of F-measure, average accuracy and weighted average F-measure. |
| **CONCLUSION** | Highest F-measure varies within 69–76%, while both highest average classification accuracy and highest weighted average F-measure values vary within 75–83% depending on the project. Their experiments show that random forests perform best, while SVM with certain kernels also achieve high performance. |

**TABLE 4**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Security Versus Performance Bugs: A Case Study on Firefox** |
| **AUTHORS** | ShahedZaman, Bram Adams, Ahmed E. Hassan |
| **YEAR** | 2011 |
| **INTRODUCTION** | A good understanding of the impact of different types of bugs on various project aspects is essential to improve software quality research and practice. For instance, we would expect that security bugs are fixed faster than other types of bugs due to their critical nature. However, prior research has often treated all bugs as similar when studying various aspects of software quality (e.g., predicting the time to fix a bug), or has focused on one particular type of bug (e.g., security bugs) with little comparison to other types. In this paper, we study how different types of bugs (performance and security bugs) differ from each other and from the rest of the bugs in a software project. Through a case study on the Firefox project, we find that security bugs are fixed and triaged much faster, but are reopened and tossed more frequently. Furthermore, we also find that security bugs involve more developers and impact more files in a project. Our work is the first work to ever empirically study performance bugs and compare it to frequently-studied security bugs. Our findings highlight the importance of considering the different types of bugs in software quality research and practice. |
| **METHODS** | First, they extract the necessary data from the bug repository (Bugzilla) and source code repository (CVS). Then, they classify the bug reports related to performance and security. Using CVS data, they also identify the bug fix information. For every type of bug, we calculate several metrics, then statistically compare the metrics across the types of bugs (performance, security and other). |
| **CONCLUSION** | They found that security bugs in Firefox behave differently than other. Security bugs require more developers with more experience, but need less triage time and are fixed faster than others. At the same time, security bug fixes are more complex than the fixes of performance and other bugs, and are reopened and tossed more than performance bugs. Similar to security bugs, performance bugs require more experienced developers to fix, and more developers are involved in performance bugs than other bugs. In terms of bug triage time, performance bugs are not different from any other bugs. Furthermore, fixing a performance bug requires changes in more files, so performance bug fixers will need good knowledge of the system. |

**TABLE 5**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **It’s Not a Bug, It’s a Feature: How Misclassification Impacts Bug Prediction** |
| **AUTHORS** | Kim Herzig, Sascha Just and Andreas Zeller |
| **YEAR** | 2013 |
| **INTRODUCTION** | In a manual examination of more than 7,000 issue reports from the bug databases of five open-source projects, they found 33.8% of all bug reports to be misclassified—that is, rather than referring to a code fix, it resulted in a new feature, an update to documentation, or an internal refactoring. This misclassification introduces bias in bug prediction models, confusing bugs and features: On average, 39% of files marked as defective actually never had a bug. They discuss the impact of this misclassification on earlier studies and recommend manual data validation for future studies. |
| **METHODS** | To validate the issue categories contained in the project’s bug databases, they manually inspected all 7,401 issue reports and checked if the type of each report reflects the maintenance task the developer had to perform in order to fix the corresponding issue. For their manual inspections, they used (a) the issue report itself, (b) all the attached comments and discussions, as well as (c) the code change that was applied to the source code.They analyzed code changes if and only if neither the issue report nor its comments clarified the underlying problem of the reported issue. Each issue report was then categorized into one of eleven different issue report categories. |
| **CONCLUSION** | First and foremost, automated quantitative analysis should always include human qualitative analysis of the input data—and of the findings. Approaches relying on bug datasets should be preceded by a careful manual validation of data quality; at least of a significant sample. Data quality should be discussed as a threat to validity. Secondly,bug prediction models trained and evaluated on biased data sets are threatened to predict changes instead of bugs. Filtering out non-bugs when estimating code quality might even improve results. The categorization of bug reports is dependent on the perspective of the observer. Approaches using bug data sets should validate whether the perspective of the prediction model matches the perspective of the bug creator. |

**TABLE 6**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **An Automated Approach for Software Bug Classification** |
| **AUTHORS** | Neelofar, Prof Dr. Muhammad YounusJaved, HufsaMohsin |
| **YEAR** | 2012 |
| **INTRODUCTION** | Open source projects for example Eclipse and Firefox have open source bug repositories. User reports bugs to these repositories. Users of these repositories are usually non-technical and cannot assign correct class to these bugs. Triaging of bugs, to developer, to fix them is a tedious and time consuming task. Developers are usually expert in particular areas. For example, few developers are expert in GUI and others are in java functionality. Assigning a particular bug to relevant developer could save time and would help to maintain the interest level of developers by assigning bugs according to their interest. However, assigning right bug to right developer is quite difficult for tri-ager without knowing the actual class, the bug belongs to. In this research, they have classified the bugs in different labels on the basis of summary of the bug. Multinomial Naïve Bayes text classifier is used for classification purpose. For feature selection, Chi-Square and TFIDF algorithms were used. Using Naïve Bayes and Chi- square, they get average of 83 % accuracy. |
| **METHODS** | When the bug is first reported to repository, it is submitted to their proposed system as shown in Fig. System extracts all the terms in these reports using bag of words approach. The vocabulary is that of extremely high dimensionality and thus numbers of features are reduced by using chi-square algorithm. These features are used for training of classification algorithm which is then used for classification of bug reports. The classification algorithm used in proposed system is multinomial Naïve Bayes. |
| **CONCLUSION** | Results are obtained on the basis of prediction accuracy. Prediction Accuracy is defined as “*Ratio of the bug reports with correct class to the total number of bug reports*”. For feature extraction TFIDF and Chi Square algorithms are used. Experimental results showed that Chi Square gives better results in case of bug classification from open source bug repositories. Chi Square gives higher accuracy as compared to TFIDF with same testing to training ratio. It clearly shows that prediction accuracy increases as training to testing ratio increases. Highest accuracy is obtained when this ratio is 1:11. Increasing the training to testing ratio although increases the prediction accuracy but execution time of algorithm increases as well. So, increasing the training vocabulary data beyond a certain limit is not feasible in real time applications. In this research an automated system for classifying software bugs is devised, using multinomial Naïve Bayes text classifier. Chi Square and TFIDF are used for feature selection. Maximum of 86% prediction accuracy is obtained. |

**TABLE 7**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Fine-grained Incremental Learning and Multi-feature Tossing Graphs to Improve Bug Triaging** |
| **AUTHORS** | Pamela Bhattacharya, IulianNeamtiu |
| **YEAR** | 2010 |
| **INTRODUCTION** | Assigning a bug to a potential developer, also known as bug triaging, is a labour-intensive, time-consuming and faultprone process if done manually. Moreover, bugs frequently get reassigned to multiple developers before they are resolved, a process known as bug tossing. Researchers have proposed automated techniques to facilitate bug triaging and reduce bug tossing using machine learning-based prediction and tossing graphs. While these techniques achieve good prediction accuracy for triaging and reduce tossing paths, they are vulnerable to several issues: outdated training sets, inactive developers, and imprecise, singleattribute tossing graphs. In this paper they improve triaging accuracy and reduce tossing path lengths by employing several techniques such as refined classification using additional attributes and intra-fold updates during training, a precise ranking function for recommending potential tosses in tossing graphs, and multi-feature tossing graphs. They validate their approach on two large software projects, Mozilla and Eclipse, covering 856,259 bug reports and 21 cumulative years of development. Their improvements have the potential to significantly reduce the bug fixing effort, especially in the context of sizable projects with large numbers of testers and developers. |
| **METHODS** | Algorithm consists of four stages, (1) initial classifier training and building the tossing graphs, (2) predicting potential developers, using the classifier and tossing graphs, (3) measuring prediction accuracy, (4) updating the training sets using the bugs which have been already validated, re-running the classifier and updating the tossing graphs. They iterate these four steps until all bugs have been validated. Tossing graphs are built using tossing probabilities derived by analyzing bug tossing histories determine potential tosses as follows: if developer A has tossed more bugs to developer B than A has tossed to D, in the future, when A cannot resolve a bug, the bug will be tossed to B, hence tossing probabilities determine tosses. Although tossing graphs reveal tossing probabilities among developers, they should also contain information about which classes of bugs were passed from one developer to another; they use multi-feature tossing graphs to capture this information |
| **CONCLUSION** | They employed three novel extensions to prior triaging approaches and showed that they could achieve higher prediction accuracy in recommending potential developers and higher reductions in tossing path lengths. In particular, they show how intra-fold updates are beneficial for achieving higher prediction accuracy in bug triaging when using classifiers in isolation. They also show that developer recommendation is improved when classifying developers based on the product-component a bug belongs to,in addition to the bug types they have fixed in the past. They validated their approach on two large, long-lived opensource projects; in the future, they plan to test how their current model generalizes to projects of different scale and lifespan. They also intend to test their approach on proprietary software. Since classifiers are often domain-based, they plan to investigate how different classifiers, and different feature sets would affect prediction accuracy. |

**TABLE 8**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Deep Triage: Exploring the Effectiveness of Deep Learning for Bug Triaging** |
| **AUTHORS** | Senthil Mani, AnushSankaran, Rahul Aralikatte. |
| **YEAR** | 2019 |
| **INTRODUCTION** | For a given software bug report, identifying an appropriate developer who could potentially fix the bug is the primary task of bug triaging. Automatic bug triaging is formulated as a classification problem, which takes the bug title and description as the input, and maps it to one of the available developers. A major challenge in doing this is that the bug description usually contains a combination of unstructured text, code snippets, and stack traces making the input data highly noisy. The existing bag-of-words (BOW) models do not consider the semantic information in the unstructured text. In this research, they propose a novel bug report representation using a deep bidirectional recurrent neural network with attention (DBRNN-A) that learns the syntactic and semantic features from long word sequences in an unsupervised manner. Using attention enables the model to remember and attend to important parts of text in a bug report. For training the model, theyuse unfixed bug reports (which constitute about 70% of bugs in a typical open source bug tracking system) which were ignored in previous studies. The dataset consists of 383,104 bug reports from Google Chromium, 314,388 bug reports from Mozilla Core, and 162,307 bug reports from Mozilla Firefox. When compared to other systems, they observe that DBRNN-A provides a higher rank-10 average accuracy. |
| **METHODS** | Major steps involved the proposed automated bug triaging algorithm and are explained as follows:   * A bug corpus having title, description, reported time, status, and owner is extracted from an open source bug tracking system, * Handling the URLs, stack trace, hex code, and the code snippets in the unstructured description requires customized training of the model, and hence in this research work, such content are removed in the pre-processing stage, * A set of unique words that occurred at least k-times in the corpus are extracted as the vocabulary, * The triaged bugs (D2) are used for classifier training and testing, while all the untriaged/open bugs (D1) are used to train the feature extractor (DBRNN-A), * The DBRNN-A learns a bug representation considering the bug title and description as a sequence of word tokens, * The triaged bugs (D2) are split into train and test data and 10 fold cross validation is used to remove training bias, * Feature representation for the training bug reports are extracted using the learned DB-RNN algorithm, * A supervised classifier is trained for performing developer assignment as a part of bug triaging process, * Feature representation of the testing bugs are then extracted using DBRNN-A,   Using the extracted features and the learned classifier, a probability score for every potential developer is predicted and the accuracy is computed on the test set |
| **CONCLUSION** | In this research they proposed a novel software bug report (title + description) triaging system using a Deep Bi-directional Recurrent Neural Network with Attention (DBRNN-A). The proposed system learns a paragraph level feature representation preserving the ordering of words over a longer context and also the semantic relationship. To perform experimental analysis, bug reports from three popular open source bug repositories are collected - Google Chromium (383,104), Mozilla Core (314,388), and Mozilla Firefox (162,307).Experimental results shows DBRNN-A along with the softmax classifier outperforms the other models, improving the rank-10 average accuracy in all three datasets. Further, it was studied that using only the title information for triaging significantly reduces the classification performance highlighting the importance of description. Additionally, the dataset along with its complete benchmarking protocol and implemented source code is made publicly available to increase the reproducibility of this research. |

**TABLE 9**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Generating Taxonomic Terms for Software Bug Classification by Utilizing Topic Models Based on Latent Dirichlet Allocation.** |
| **AUTHORS** | N. K. Nagwani,S. Verma,K. K. Mehta |
| **YEAR** | 2013 |
| **INTRODUCTION** | Discovering categorical (taxonomic) terms in text classification is an important and complex problem. Development of a good text classifier depends on the method of identification and generation of proper taxonomic terms. Software bug indicates improper behaviour of the functionalities given during the requirements. These bugs are tracked with the help of bug tracking systems (BTS) where the bug information is presented using several attributes out of which some important attributes are textual for example summary and description. For effective classification of the software bugs a good text classifying mechanism is required for which proper taxonomic terms are required to be identified. In this work a methodology is presented to find the taxonomic terms using Latent Dirichlet Allocation (LDA) for software bug classification. |
| **METHODS** | **Method** - The methodology of the proposed work is composed of six major steps, which are mentioned and explained as under.   * **Retrieving the software bugs***-* Bug repositories are consisting of thousands of bugs present in the software however for the present work random samples are retrieved from the bug repositories to generate the categorical terms for the software bug classification. After retrieval of software bugs parsing is performed for extracting the various attributes of a software bug like summary, description etc. * **Performing text pre-processing** *-* The text pre-processing is performed to eliminate the stop words and to extract the stems of the information present in the textual attributes of the software bugs. The purpose of creating the clusters is two folds, first it will create the groups of similar bugs and second the topic modelling will be carried out for each individual cluster, which will make sure that the topic terms from each group is considered for taxonomic terms. * **Applying LDA on bug clusters to discover topics** *-* After the bug clusters are created, LDA is applied to the individual cluster by taking the collection of textual attributes (summary and description) of bugs belonging to that particular cluster. Ten topics are generated using ten terms present in each topic. Once the topics are generated for each cluster using LDA, then individual term is captured and its frequency is counted for occurrences in the topic of all the clusters. * **Filtering the most dominating terms from the topic terms** *-* Based on the frequencies of the topics terms present in the topics of the bug clusters, the topic terms with maximum frequencies are filtered as the taxonomic terms. |
| **CONCLUSION** | An effective approach for generating the taxonomic terms for software bug classification using LDA is presented in this work. The implementation of the proposed work is carried out in open source technologies java and mallet and dataset from open source software bug repositories is taken. These terms can be utilized efficiently for classifying the software bugs. The future work related to the present work could be development of a text classifier by utilizing the taxonomic terms present in the work for classification of software bugs and comparing the developed classifier with other existing bug classification schemes. |

**TABLE 10**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Recognizing Software Bug-Specific Named Entity in Software Bug Repository** |
| **AUTHORS** | Cheng Zhou, Bin Li, Xiaobing Sun, HongjingGuo. |
| **YEAR** | 2018 |
| **INTRODUCTION** | The rich information in the bug repository provides the possibility of establishment of entity-centric knowledge bases to help understand and fix the bugs. However, existing named entity recognition (NER) systems deal with text that is structured, formal, well written, with a good grammatical structure and few spelling errors, which cannot be directly used for bug-specific named entity recognition. For bug data, they are free-form texts, which include a mixed language studded with code, abbreviations and software-specific vocabularies. In this paper, they summarize the characteristics of bug entities, propose a classification method for bug entities, and build a baseline corpus on two open source projects (Mozilla and Eclipse). On this basis, they propose an approach for bug-specific entity recognition called BNER with the Conditional Random Fields (CRF) model and word embedding technique. An empirical study is conducted to evaluate the accuracy of their BNER technique, and the results show that the two designed baseline corpus are suitable for bug-specific named entity recognition, and their*BNER* approach is effective on cross-projects NER. |
| **METHODS** | **The impact of different features on BNER -** They answerby measuring the effectiveness of their*BNER* approach with different kinds of features. The study was conducted on two individual projects, respectively. There are three steps to conduct this study. First, they used the basic features (tokens and POS tags) to train the CRF model. Then, they combined the orthographic feature and gazetteer feature with the basic features. Finally, they induce the embedding feature using word vector and word cluster, respectively. For each step, the corpus was first randomly split into 10 equalized subsets in order to build the training dataset and testing dataset. Then, they used 10-fold cross validation for model training, for which one subset is used as the testing data, and the remaining nine subsets are used as training data. They repeated this process 10 times and produced a single estimation by averaging the ten results. They computed the precision, recall and F1 results, respectively. |
| **CONCLUSION** | Understanding software bug data is important for fixing the bugs in bug repository. In this paper, they proposed a semi-supervised learning approach to implement software bug-specific NER in software bug repository. Based on these characteristics, they divide the bug entity into sixteen categories and build a baseline bug corpus. Then, they develop a *BNER* approach using the linear chain CRF model with the following features: POS tags, contextual feature, orthography feature, gazetteer feature and word embedding feature using both word cluster and word vector.To demonstrate the effectiveness of their approach, they performed an empirical study on the bug repositories of two popular open: Eclipse and Mozilla. The results show that the features for CRF model are useful, and the accuracy and recall results of each category can reach over 70%, some even more than 80%. Moreover, their BNER approach can be also suitable for cross-project bug-specific named entity recognition. |

**TABLE 11**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Improved Duplicate Bug Report Identification** |
| **AUTHORS** | Yuan Tian, Chengnian Sun, and David Lo |
| **YEAR** | 2012 |
| **INTRODUCTION** | The paper answers the question: given a new bug report, classify if it as a duplicate bug report or not. In this paper, Jalbert and Weimer’s work is extended by improving the accuracy of automated duplicate bug report identification. They experimented with bug reports from Mozilla bug tracking system which were reported between February 2005 to October 2005, and find that we could improve the accuracy of the previous approach by about 160%. |
| **METHODS** | 1.)They split training data into two parts. One partis used to train the parameters of REP. To use REPto measure the similarity between two bug reports,some parameters need to be set; these parameters areweights to the different features of bug reports, i.e.,textual, ordinal, and categorical features. The otherpart is used to train the machine learning model.  2) For the latter part of the training data, each bug reportis converted to a triple using the feature engineeringstrategy.  3) They then train a machine learning model using Support Vector Machine (SVM). |
| **CONCLUSION** | Bug reporting is an uncoordinated distributed process. Thus, often there are many duplicate reports being submitted that report the same defect. To address this issue, there is a need for an automated duplicate report detection approach. There are two families of research work in this direction: report retrieval and report classification. In this work, we extend the latest stdy on report classification by Jalbert and Weimer. They extended their approach by utilizing REP which was recently proposed for report retrieval problem to measure the similarity of two bug reports. We also utilize information on the difference between product fields in two bug reports to help in identifying if two bug reports are duplicate or not. Furthermore, we define a new notion of relative similarity that help to decide if the similarity between two bug reports is significant enough. We haveperformed experiments on Mozilla’s bug reports which werereportedbetween February 2005 to October 2005.Our preliminary study has shown that our approach is effective to increase the true positive rate by 200% (from 8% to 24%) while only suffering a loss in true negative rate by 9% (from 100% to 91%). The overall harmonic mean of true positive rate and true negative rate is increased by 160% (from 14.8%to 38.62%). |

**TABLE 12**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Detecting Duplicate Bug Report Using Character N-Gram-Based Features** |
| **AUTHORS** | Ashish Sureka, Pankaj Jalote |
| **YEAR** | 2010 |
| **INTRODUCTION** | We present an approach to identify duplicate bug reports expressed in free-form text. Duplicate reports needs to be identified to avoid a situation where duplicate reports getassigned to multiple developers. Also, duplicate reports can contain complementary information which can be useful for bug fixing. Automatic identification of duplicate reports (from thousands of existing reports in a bug repository) can increase the productivity of a Triagerby reducing the amount of time a Triager spends in searching for duplicate bug reports of any incoming report. The proposed method uses character N-gram-based model for the task of duplicate bug report detection. Previous approaches are word-based whereas this study investigates the usefulness of low-level features based on characters which have certain inherent advantages (such as natural-language independence, robustness towards noisy data and effective handling of domain specific term variations) over word-based features for the problem of duplicate bug report detection. The proposed solution is evaluated on a publiclyavailable dataset consisting of more than 200 thousand bug reports from the open-source Eclipse project. The dataset consists of ground-truth (pre-annotated dataset having bug reports tagged as duplicate by the Triager). Empirical results and evaluation metrics quantifying retrieval performance indicate that the approach is effective. |
| **METHODS** | Feature extraction and similarity computation (semanticand lexical) between two bug reports is central to the problem of duplicate detection. Extracting discriminatory features and important indicators from bug description is key to the performance of any duplicate bug report detection system. In this work, we experiment with low-level character n-gram based features which can complement features proposed in previous approaches. In the following sub-section, we describe the character n-gram model and argue in support of our hypothesis (using illustrative examples from Eclipse bug database). We argue that the proposed character ngrammodel can capture important linguistic characteristics (discriminatory features) of bug reports which can play an important role for the task of duplicate identification. |
| **CONCLUSION** | This paper presents an approach to compute text similarity between two bug reports to assist a Triager in the task of duplicate bug report detection. The central idea behind the proposed approach is the application of character n-grams as low-level features to represent the title and detailed description of a bug report. The advantages of the approach are language independence as it does not require language specific pre-processing and ability to capture sub-word features which is useful in situations requiring comparison of noisy text. The approach is evaluated on a bug databaseconsisting of more than 200 thousand bug reports from opensourceEclipse project. The recall rate for the Top 50 results is 33.92% for 1100 randomly selected test cases and 61.94% for 2270 randomly selected test cases with a title to title similarity (between the master and the duplicate) of more than a pre-defined threshold of 50.. |

**TABLE 13**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Bug Report, Feature Request, or Simply Praise? On Automatically Classifying App Reviews** |
| **AUTHORS** | WalidMaalej, Hadeer Nabil |
| **YEAR** | 2015 |
| **INTRODUCTION** | App stores like Google Play and Apple AppStore have over 3 Million apps covering nearly every kind of softwareand service. Billions of users regularly download, use, and review these apps. Recent studies have shown that reviews written by the users represent a rich source of information for the app vendors and the developers, as they include information about bugs, ideas for new features, or documentation of released features. This paper introduces several probabilistic techniques to classify app reviews into four types: bug reports, feature requests, user experiences, and ratings. For this we use review metadata such as the star rating and the tense, as well as, text classification, natural language processing, and sentiment analysis techniques.We conducted a series of experiments to compare the accuracy of the techniques and compared them with simple string matching.We found that metadata alone results in a poor classification accuracy. When combined with natural language processing, the classification precision got between 70-95% while the recall between 80-90%. Multiple binary classifiers outperformed single multiclass classifiers. Our results impact the design of review analytics tools which help app vendors, developers, and users to deal with the large amount of reviews, filter critical reviews, andassign them to the appropriate stakeholders. |
| **METHODS** | The contribution of this paper is threefold. First, it introduces probabilistic techniques and heuristics for classifying the reviews based on their metadata (e.g., the star rating and text length), keyword frequencies, linguistic rules, and sentiment analysis. Second, the paper reports on an extensive study to compare the accuracy of the review classification techniques. The study data and its results serve as a benchmark for review classification. Third, we derive concrete insights into how to design and use review analytics tools for different stakeholders. |
| **CONCLUSION** | App stores provide a rich source of information for softwareprojects, as they combine technical, business, and user-relatedinformation in one place. Analytics tools can help stakeholders to deal with the large amount, the variety, and quality of the app stores data and to take the right decisions about the requirements and future releases. In this paper, we proposed and studied one functionality of app store analytics that enables the automatic classification of user reviews into bug reports, feature requests, user experiences, and ratings (i.e. simple praise or dispraise repeating the star rating). In a series of experiments, we compared the accuracy of simple string matching, textclassification, natural language processing, sentiment analysis, and review metadata to classify the reviews. We reported on several findings which can inspire the design of review analytics tools. For example, text classification should be enhanced with metadata such as the tense of the text, the star rating, thesentiment score, and the length. Moreover, stop word removal and lemmatization, two popular NLP techniques used for document classification, should be used carefully, since every word in a short informal review can informative. Overall, the precision and recall for all four classes are encouraging –ranging from 71% up to 97%. Our work helps to filter reviews of interest for certain stakeholders as developers, analysts, and other users. Complementary within-app analytics such as the feature extraction, opinion mining, and the summarization of the reviews, will make app store data more useful for softwareand requirements engineering decisions. |

**TABLE 14**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Automatic bug triage using text categorization** |
| **AUTHORS** | Murphy, G., &Cubranic, D. |
| **YEAR** | 2004 |
| **INTRODUCTION** | Bug triage, deciding what to do with an incoming bug report, is taking up increasing amount of developer resources in large open-source projects. In this paper, we propose to apply machine learning techniques to assist in bug triage by using text categorization to predict the developer that should work on the bug based on the bug’s description. We demonstrate our approach on a collection of 15,859 bug reports from a large open-source project. Our evaluation shows that our prototype, using supervised Bayesian learning, can correctly predict 30% of the report assignments todevelopers. |
| **METHODS** | 1. If the report was *resolved* by the assigned-to developer, the report is labelled by his or her class regardless of who the submitter was or what the report’s *resolution*was (e.g., *fixed, duplicate, invalid, later*, etc.). This is clearly the case of a developer who was in charge ofthe report and who has completed processing it.  2. If the report was *resolved* by someone other than theassigned-to developer, but not by the person who submitted it, we label the report with the class of the developer who marked it resolved. The reasoning is that whoever made the decision to resolve the report is the person to whom it should have been assigned all along.  3. If the report was *resolved* as *fixed*, regardless of who the resolver was, we assume that this is the developerwho implemented the fix and label the report with the class of that developer, as this is probably the personwho had done the real work on the report. This rulecovers the frequent case where an Eclipse developer files a report, which is then assigned to somebody else or a sub-team alias by default, and then later implements the fix himself.  4. If the report was *resolved* as non-fixed (i.e., with resolution *duplicate*, *invalid*, etc.) by the person who submitted it, and who was not also assigned to it, the report is labelled with the class of the first person who responded to the reporter. This handles the many casesof a submitter throwing the report away after being informedthat it is a feature and not a bug, or after beingprompted by a developer for details of his or her setupand discovering that the bug does not exist any more.We choose the first responder to the report rather thanthe assigned-to person for reasons outlined above.  5. If the report was *resolved* as non-fixed by the submitter who was not the assigned-to developer, and nobody responded, we assume that the report was submitted in error—for example, not knowing the proper operation of Eclipse—and that the mistake was caught by the submitter before anyone could react. These reports are removed from the training set, as they cannot be reliably labelled.  6. If the report was not *resolved*, we label it with the class of the most recent assigned-to developer. |
| **CONCLUSION** | In this paper, we described an application of supervisedmachine learning using a naive Bayes classifier toautomatically assign bug reports to developers. We evaluatedour approach on bug reports from a large open-sourceproject, Eclipse.org, achieving 30% classification accuracywith current prototype. We believe that the system could beeasily incorporated into current bug-handling procedures todecrease the resources currently devoted to bug triage. Newbug reports would be automatically assigned to the developerpredicted to be the most appropriate to the content.Mispredictions could be handled in a light-weight fashionby their assignee, “bouncing” them to a dedicated triagerfor human inspection and classification. Clearly, even theclassification accuracy we can currently achieve, would significantlylighten the load that the triagers face under thepresent conditions. |

**TABLE 15**

|  |  |
| --- | --- |
| **TITLE OF THE PAPER** | **Who Should Fix This Bug?** |
| **AUTHORS** | Anvik, J., Hiew, L., & Murphy, G. C |
| **YEAR** | 2006 |
| **INTRODUCTION** | Open source development projects typically support an open bug repository to which bothdevelopers and users can report bugs. The reports that appear in this repository must be triaged to determine if the report is one which requires attention and if it is, which developer will be assigned theresponsibility of resolving the report. Large open source developments are burdened by the rate at which new bug reports appear in the bug repository. In this paper, we present a semi-automated approach intended to ease one part of this process, the assignment of reports to a developer. Our approach applies a machine learning algorithm to the open bug repository to learn the kinds of reports each developer resolves. When a new report arrives, the classifier produced by the machine learning technique suggests a small number of developers suitable to resolve the report. With this approach, we have reached precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively. We have also applied our approach to the gcc open source development with less positive results. We describe the conditions under which the approach is applicable and also report on the lessons we learned about applying machine learning to repositories used in open source development. |
| **METHODS** | Our approach consists of four steps:  1. characterizing bug reports,  2. assigning a label to each report,  3. choosing reports to train the supervised machine learning algorithm, and  4. applying the algorithm to create the classifier for recommending assignments. |
| **CONCLUSION** | In this paper, we have presented an approach to semi-automating the assignment of a bug report to a developer with the appropriate expertise to resolve the report. Our approach uses a supervised machine learning algorithm that is applied to information in the bug repository. For the Eclipse and Firefox projects, we are able to achieve precision rates of over 50%, reaching 64% on one recommendation for Firefox. Our results for the gcc project were far worse, where we achieved a precision rate for one recommendation of only 6% because of characteristics of the project, such as one developer dominating the report resolution process. In addition to presenting our approach and results, we have presented an in-depth analysis of the application of machine learning to this problem and we have reported on lessons learned in trying to make use of data in the bug repository. We believe that our approach shows promise for improving the bug assignment problem for open source software developments. Our future plans include an empirical study of the use of the approach by bug triagers on an open source system, an investigation of additional sources of information, and a prescriptive means for determining when the approachmay be applicable. |

* 1. **INTEGRATED SUMMARY OF THE LITERATURE STUDIED**

Open source projects for example Eclipse and Firefox have open source bug repositories. User reports bugs to these repositories. Users of these repositories are usually non-technical and cannot assign correct class to these bugs. Triaging of bugs, to developer, to fix them is a tedious and time consuming task. Developers are usually expert in particular areas. For example, few developers are expert in GUI and others are in java functionality. Assigning a particular bug to relevant developer could save time and would help to maintain the interest level of developers by assigning bugs according to their interest. However, assigning right bug to right developer is quite difficult for tri-ager without knowing the actual class, the bug belongs to.Automatic bug triaging is formulated as a classification problem, which takes the bug title and description as the input, and maps it to one of the available developers. A major challenge in doing this is that the bug description usually contains a combination of unstructured text, code snippets, and stack traces making the input data highly noisy. The existing bag-of-words (BOW) models do not consider the semantic information in the unstructured text.

Results are obtained on the basis of prediction accuracy. Prediction Accuracy is defined as “*Ratio of the bug reports with correct class to the total number of bug reports*”. For feature extraction TFIDF and Chi Square algorithms are used. Experimental results showed that Chi Square gives better results in case of bug classification from open source bug repositories. Chi Square gives higher accuracy as compared to TFIDF with same testing to training ratio. It clearly shows that prediction accuracy increases as training to testing ratio increases. Highest accuracy is obtained when this ratio is 1:11. Increasing the training to testing ratio although increases the prediction accuracy but execution time of algorithm increases as well. So, increasing the training vocabulary data beyond a certain limit is not feasible in real time applications. In this research an automated system for classifying software bugs is devised, using multinomial Naïve Bayes text classifier. Chi Square and TFIDF are used for feature selection. Maximum of 86% prediction accuracy is obtained.Bug fixing is an important activity for almost all software organizations. The ability to predict bug-fixing time can help estimate maintenance effort and improve project management.

**CHAPTER-3**

**REQUIREMENT ANALYSIS AND SOLUTION APPROACH**

**DESCRIPTION OF THE PROJECT**

1. **PURPOSE**

In the current era when a large volume of bugs occur in different software packages,  bug triaging has become one of the most important issues, along with the great concern of prioritizing issues . Bugs affect the software performance,sometimes defeating the services and purpose served by the software and causing a severe challenge to the developers. Issue labelling and prioritization process is usually performed manually which makes it error-prone and labor intensive. It relies heavily on the triager judgment and experience. Many bug reports have been assigned incorrect priority levels and many of them are usually left blank since bug prioritization needs a deep knowledge of bug reports. Wrong assignments of priority levels may lead to utilizing resources ineffectively (e.g., wasting time and effort by fixing unimportant bugs first).  Even though, little work has been done to predict other characteristics such as severity and priority of bug reports. In this project, we investigate whether we can accurately predict the priority of a reported bug by using several features such as the textual description of bug reports.

1. **SCOPE**

The purpose of this study is to apply ensemble methods, to the study of issue reports, classifying issue reports according to summary and description features in report, investigating correlation between issue type,mean time to repair and summary analysis, and considering applicability of the different ensemble techniques as a useful computational method in the study of reports. The first consideration for doing such a study is to find distribution of issue reports. Ensemble techniques are a better alternative when individual classifiers are not able to produce optimal results.

1. **PRODUCT PERSPECTIVE**

**Ensemble learning** is the process by which multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular [computational intelligence](http://www.scholarpedia.org/article/Computational_intelligence) problem. Ensemble learning is primarily used to improve the (classification, prediction, function approximation, etc.) performance of a model.

Within the field of machine learning, there are two main types of tasks: supervised, and unsupervised. In Supervised learning, you train the machine using data which is well labelled. It means some data is already tagged with the correct answer. Unsupervised learning is a machine learning technique, where you do not need to supervise the model. Instead, you need to allow the model to work on its own to discover information. It mainly deals with the unlabelled data.

Ensemble methods **decreasevariance** (bagging), **bias** (boosting), or **improve predictions** (stacking).They can be divided into two groups:

* Sequential - Ensemble methods where the base learners are generated sequentially (e.g. AdaBoost). The basic motivation of sequential methods is to **exploit the dependence between the base learners.** The overall performance can be boosted by weighing previously mislabeled examples with higher weight.
* Parallel - Ensemble methods where the base learners are generated in parallel (e.g. Random Forest).The basic motivation of parallel methods is to **exploit independence between the base learners** since the error can be reduced dramatically by averaging.

In order for ensemble methods to be more accurate than any of its individual members, the base learners have to be as accurate as possible and as diverse as possible.

We use heterogeneous learners, i.e. learners of different types, leading to heterogeneous ensembles.

**REQUIREMENT ANALYSIS**

JIRA is a tool developed by Australian Company Atlassian. It is used for bug tracking, issue tracking, and project management. The basic use of this tool is to track issue and bugs related to your software and [Mobile](https://www.guru99.com/mobile-testing.html) apps. It is also used for project management. The JIRA dashboard contains issue reports with the following key features are listed below.

1. Issue Types

Issue Type displays all types of items that can be created and tracked via JIRA. JIRA Issues are classified under various forms like new feature, sub-task, bug, etc.

1. Issue Attributes

Issue Attributes encompasses

* + - * 1. Statuses
        2. Resolutions
        3. Priorities

**Status:** Different statuses are used to indicate the progress of a project like To do, InProgress, Open, Closed, Reopened, and Resolved. Likewise, we have resolutions and priorities, in resolution it again tells about the progress of issue like Fixed, Won't fix, Duplicate, Incomplete, Cannot reproduce, Done also we can set the priorities of the issue whether an issue is critical, major, minor, blocker and Trivial.

**Summary:**Provides high level context of the Project, a glimpse into issues, versions and the activity stream

**Description:** Brief explanation and insights on issues with Project implementation.

Project Name: Name of Project corresponding to which the Issue Report Type has been generated.

Project Description:Project functionality and implementation details

Created:Time and Date of Issue Report creation

Updated:Time and Date of Issue Report updation

Resolved:Time and Date of Issue Report resolution

Other key features include:

Issue Key

Issue Id

Project key

Project Type Lead

Assignee

**Software requirements:**

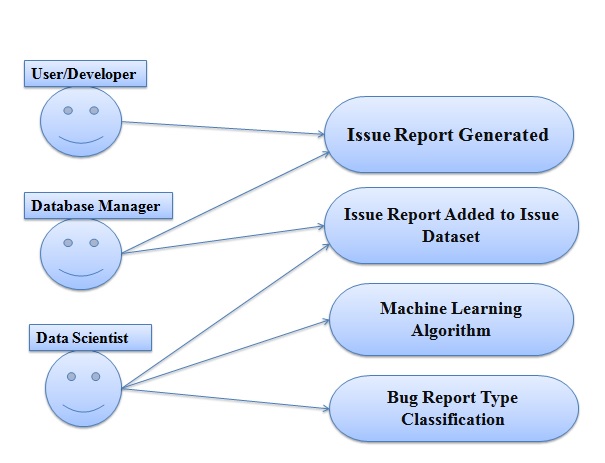
* Operating Systems :MacOs and Windows
* Language: Python
* Tool: Anaconda - Jupyter
* Python libraries:
  + - 1. From datetime import datetime
      2. From sklearn.preprocessing import MinMaxScaler
      3. From sklearn.metrics import classification\_report
      4. From sklearn.metrics import confusion\_matrix
      5. Fromsklearn.feature\_extraction.textimportTfidfVectorizer
      6. fromsklearn.feature\_extraction.textimportCountVectorizer
      7. fromsklearn.naive\_bayesimportBernoulliNB
      8. fromsklearn.naive\_bayesimportMultinomialNB
      9. fromsklearn.ensembleimportRandomForestClassifier
      10. from Sklearn import linear\_model
      11. fromsklearn.ensembleimportVotingClassifier
      12. fromsklearn.calibrationimportCalibratedClassifiercV
      13. fromsklearn.ensembleimportGradientBoostingClassifier

**CHAPTER 4**

**MODELING AND IMPLEMENTATION DETAILS**

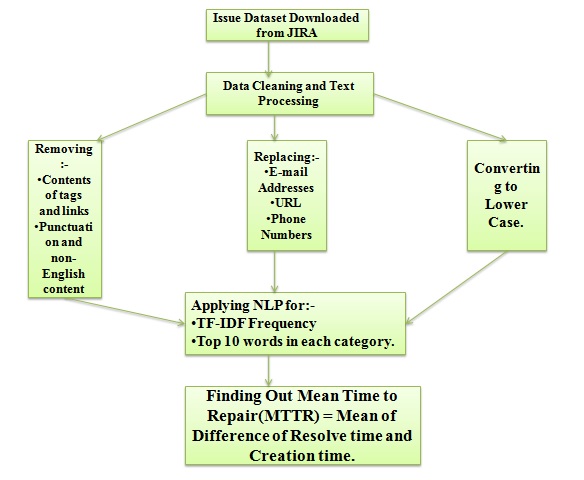
**4.1) DESIGN DIAGRAMS**

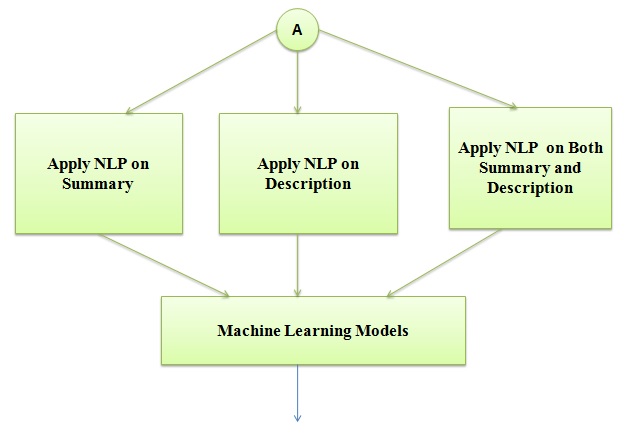
**4.1.1) USE CASE DIAGRAMS**

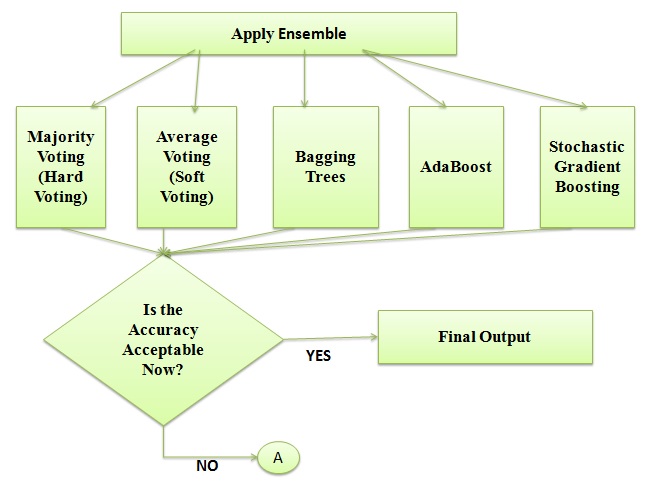
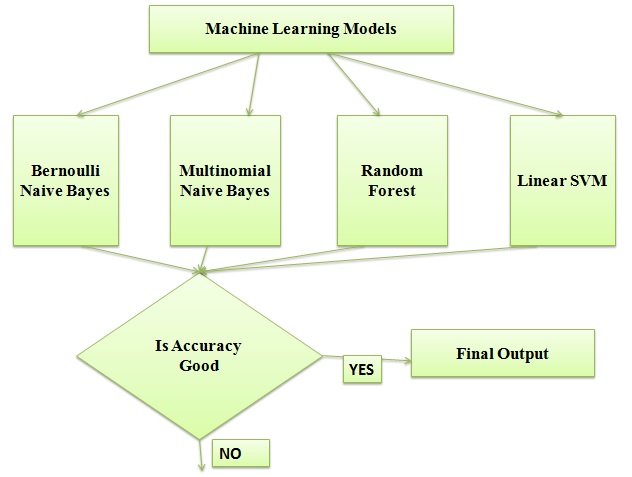
****

**Fig - 5**

**4.1.2)CLASS DIAGRAMS/CONTROL FLOW DIAGRAMS**

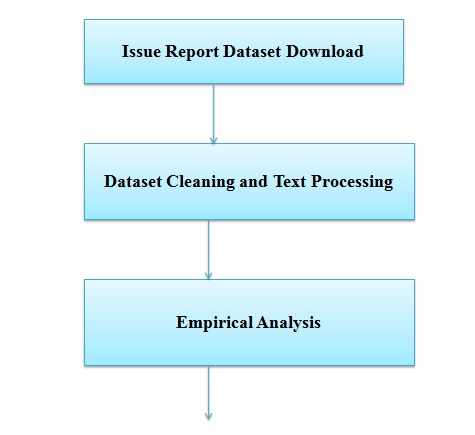
****

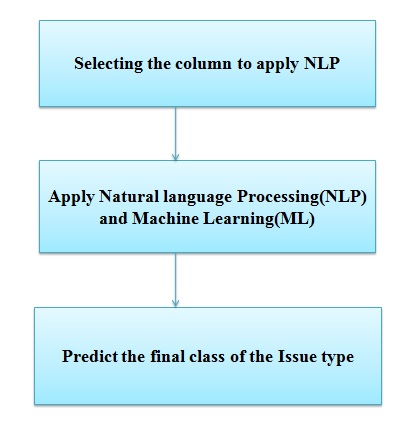
****

****

**Fig - 6**

**4.1.3)ACTIVITY DIAGRAM**





**Fig - 7**

**4.2) IMPLEMENTATION**

After cleaning and basic text processing we had refined dataset consisting **7191 rows and 11 columns**. Out of which the two major ones are –

* Summary – Containing the Summary of the Bug report generated.
* Description – Containing the complete description of the bug, ranging from where the software crashed to where the developer faced problems.

Two approaches were used –

1. Applying NLP and TFIDF vectorizer in individual fields of Summary and Description and further analysis.
2. Combining the Summary and Description, and then applying TFIDF vectorizer on that particular feature.

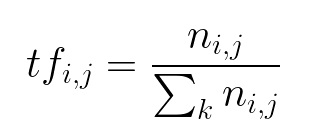
So we combined the columns of Description and summary and then applied our NLP.

We started by having the whole of the corpus in a variable. Machine learning algorithms cannot work with raw text directly. Rather, the text must be converted into vectors of numbers. In natural language processing, a common technique for extracting features from text is to place all of the words that occur in the text in a bucket. This approach is called a **bag of words**. It’s referred to as a “bag” of words because any information about the structure of the sentence is lost. Next, created a dictionary of words and their occurrence for each document in the corpus (collection of documents). The problem with the bag of words approach is that it doesn’t account for noise. In other words, certain words are used to formulate sentences but do not add any semantic meaning to the text. For example, the most commonly used word in the english language is the which represents 7% of all words written or spoken. You couldn’t make deduce anything about a text given the fact that it contains the word **the**. On the other hand, words like **good** and **awesome** could be used to determine whether a rating was positive or not.

Often times, when building a model with the goal of understanding text, you’ll see all of stop words being removed. Another strategy is to score the relative importance of words using TF-IDF.

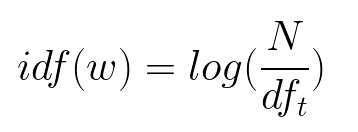
**Term Frequency (TF)**

The number of times a word appears in a document divided by the total number of words in the document. Every document has its own term frequency.

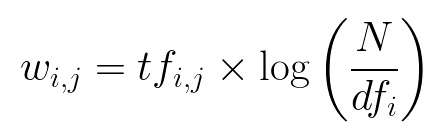


## **Inverse Data Frequency (IDF)**

The log of the number of documents divided by the number of documents that contain the word w. Inverse data frequency determines the weight of rare words across all documents in the corpus.



Lastly, the TF-IDF is simply the TF multiplied by IDF.



With the data pre-processed, we developed the models. When it comes to developing machine learning models (and in our particular case, classifiers), we need to firstly train them on the labelled training data to learn from and then use the test data-set to make predictions. So to do that, we will proceed with splitting our existing data-set into training and test data.

When text data is pre-processed, the issue of curse of dimensionality usually appears i.e. data becomes highly multi-dimensional with lots of features ranging in thousands. Not all of those features are helpful and also it adversely affects the performance of classifiers as well so following the best practices, we opted for best-in-class feature extraction methods and also applied feature selection techniques so as to compile only those features that will contribute in this prediction problem. For model development, we compared the following set of machine learning algorithms:

1. Bernoulli Naive Bayes
2. Multinomial Naive Bayes
3. Random Forests
4. Linear SVM

**Model Used**

**Naive Bayes** - Naive Bayes is one of the most widely used classification algorithm in text mining applications. Based on Bayes theorem, this model makes the assumption that all the features are independent of each other and uses the probabilities of each attribute belonging to each class to make a prediction. The condition of independence may not be valid in many circumstances but as a base line model, it’s a good starting point to test its performance on the provided data. There are two forms of Naive Bayes –

1. Bernoulli Naïve Bayes
2. Multinomial Naïve Bayes

**Random Forest Classifier** - Random forests or random decision forests are an [ensemble learning](https://en.wikipedia.org/wiki/Ensemble_learning) method for [classification](https://en.wikipedia.org/wiki/Statistical_classification), [regression](https://en.wikipedia.org/wiki/Regression_analysis) and other tasks that operates by constructing a multitude of [decision trees](https://en.wikipedia.org/wiki/Decision_tree_learning) at training time and outputting the class that is the [mode](https://en.wikipedia.org/wiki/Mode_(statistics)) of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of [overfitting](https://en.wikipedia.org/wiki/Overfitting) to their [training set](https://en.wikipedia.org/wiki/Test_set).



**Fig - 8**

# ****Linear SVM****

In [machine learning](https://en.wikipedia.org/wiki/Machine_learning), support-vector machines (SVMs, also support-vector networks[[1]](https://en.wikipedia.org/wiki/Support-vector_machine#cite_note-CorinnaCortes-1)) are [supervised learning](https://en.wikipedia.org/wiki/Supervised_learning) models with associated learning [algorithms](https://en.wikipedia.org/wiki/Algorithm) that analyze data used for [classification](https://en.wikipedia.org/wiki/Statistical_classification) and [regression analysis](https://en.wikipedia.org/wiki/Regression_analysis). Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-[probabilistic](https://en.wikipedia.org/wiki/Probabilistic_classification) [binary](https://en.wikipedia.org/wiki/Binary_classifier) [linear classifier](https://en.wikipedia.org/wiki/Linear_classifier) (although methods such as [Platt scaling](https://en.wikipedia.org/wiki/Platt_scaling) exist to use SVM in a probabilistic classification setting). An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall.

Stochastic Gradient Descent (SGD) is a one of the most efficient approaches used in linear classifiers under convex loss functions such as (linear) Support Vector Machines. It has proven to perform well in in large-scale and sparse machine learning problems.

So far we see that our relative accuracy touches around 63-65%, in order to increase the overall accuracy we applied the ensemble learning technique.

**Ensemble Learning** - [Supervised learning](https://en.wikipedia.org/wiki/Supervised_learning) algorithms perform the task of searching through a hypothesis space to find a suitable hypothesis that will make good predictions with a particular problem. Even if the hypothesis space contains hypotheses that are very well-suited for a particular problem, it may be very difficult to find a good one. Ensembles combine multiple hypotheses to form a (hopefully) better hypothesis. The term *ensemble* is usually reserved for methods that generate multiple hypotheses using the same base learner. The broader term of *multiple classifier systems* also covers hybridization of hypotheses that are not induced by the same base learner.

Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation. Fast algorithms such as [decision trees](https://en.wikipedia.org/wiki/Decision_tree_learning) are commonly used in ensemble methods (for example, [random forests](https://en.wikipedia.org/wiki/Random_forest)), although slower algorithms can benefit from ensemble techniques as well.

An ensemble is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensembles can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to [over-fit](https://en.wikipedia.org/wiki/Overfitting) the training data more than a single model would, but in practice, some ensemble techniques (especially [bagging](https://en.wikipedia.org/wiki/Bootstrap_aggregating)) tend to reduce problems related to over-fitting of the training data.

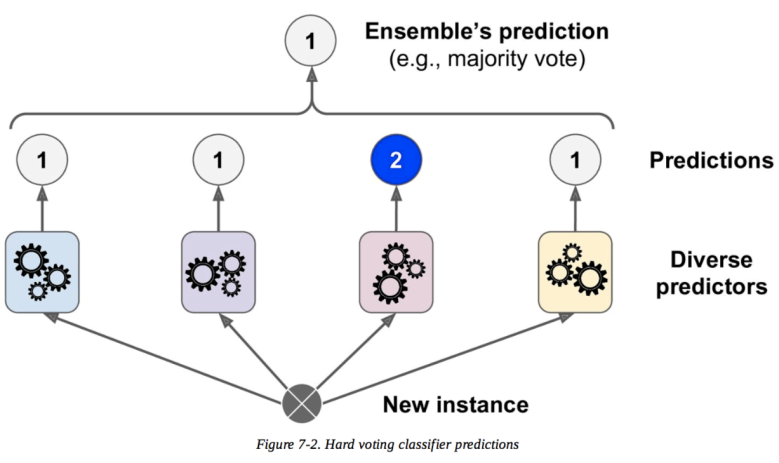
Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms (like random decision trees) can be used to produce a stronger ensemble than very deliberate algorithms (like entropy-reducing decision trees). Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to *dumb-down* the models in order to promote diversity.

Ensemble methods we have used –

* Majority Voting(Hard Voting)
* Average Voting(Soft Voting)
* Bagging of Decision Trees
* AdaBoost
* Stochastic Gradient Descent Boosting

You can train your model using diverse algorithms and then ensemble them to predict the final output. Say, you use a Random Forest Classifier, SVM Classifier, Linear Regression etc.; models are pitted against each other and selected upon best performance by voting using the Voting Classifier Class from sklearn ensemble.

Majority Voting - Every model makes a prediction (votes) for each test instance and the final output prediction is the one that receives more than half of the votes. If none of the predictions get more than half of the votes, we may say that the ensemble method could not make a stable prediction for this instance.



**Fig - 9**

**Soft Voting** - Soft Voting can only be done when all your classifiers can calculate probabilities for the outcomes. Soft voting arrives at the best result by averaging out the probabilities calculated by individual algorithms.

**Bagged Decision Tree** - Bootstrap aggregating, also called bagging (from bootstrap aggregating), is a [machine learning ensemble](https://en.wikipedia.org/wiki/Ensemble_learning) [meta-algorithm](https://en.wikipedia.org/wiki/Meta-algorithm) designed to improve the stability and accuracy of [machine learning](https://en.wikipedia.org/wiki/Machine_learning) algorithms used in [statistical classification](https://en.wikipedia.org/wiki/Statistical_classification) and [regression](https://en.wikipedia.org/wiki/Regression_analysis). It also reduces [variance](https://en.wikipedia.org/wiki/Variance) and helps to avoid [overfitting](https://en.wikipedia.org/wiki/Overfitting). Although it is usually applied to [decision tree](https://en.wikipedia.org/wiki/Decision_tree_learning) methods, it can be used with any type of method. Bagging is a special case of the [model averaging](https://en.wikipedia.org/wiki/Ensemble_learning) approach.

**Boosting** - Boosting is a method of converting weak learners into strong learners. In boosting, each new tree is a fit on a modified version of the original data set. The gradient boosting algorithm (GBM) can be most easily explained by first introducing the AdaBoost Algorithm. The AdaBoost Algorithm begins by training a decision tree in which each observation is assigned an equal weight. After evaluating the first tree, we increase the weights of those observations that are difficult to classify and lower the weights for those that are easy to classify. The second tree is therefore grown on this weighted data. Here, the idea is to improve upon the predictions of the first tree. Our new model is therefore Tree 1 + Tree 2. We then compute the classification error from this new 2-tree ensemble model and grow a third tree to predict the revised residuals. We repeat this process for a specified number of iterations. Subsequent trees help us to classify observations that are not well classified by the previous trees. Predictions of the final ensemble model is therefore the weighted sum of the predictions made by the previous tree models.

**Adaboost**- First you select a base classifier which makes predictions on the given set. Note down the misclassified instances. The weights of the misclassified instances are increased. A second classifier is trained on the training set with updated weights.

In simple terms, Run a Classifier and make predictions. Run another classifier to fit the previously misclassified instances and make predictions. Repeat until all/most of the training instances are fitted.

Instead of a Decision Tree, AdaBoost uses a Decision Stump which is a decision tree with max\_depth = 1, i.e., Tree of single decision node and two leaf nodes. The n\_estimators parameters in AdaBoost sets the number of Decision Stumps.

**Stochastic Gradient Boosting** -Gradient boosting is a [machine learning](https://en.wikipedia.org/wiki/Machine_learning) technique for [regression](https://en.wikipedia.org/wiki/Regression_(machine_learning)) and [classification](https://en.wikipedia.org/wiki/Classification_(machine_learning)) problems, which produces a prediction model in the form of an [ensemble](https://en.wikipedia.org/wiki/Ensemble_learning) of weak prediction models, typically [decision trees](https://en.wikipedia.org/wiki/Decision_tree_learning). It builds the model in a stage-wise fashion like other [boosting](https://en.wikipedia.org/wiki/Boosting_(machine_learning)) methods do, and it generalizes them by allowing optimization of an arbitrary [differentiable](https://en.wikipedia.org/wiki/Differentiable_function) [loss function](https://en.wikipedia.org/wiki/Loss_function). Gradient Boosting also works with successive predictive models added to the ensemble. Instead of updating the weights of the training instances like AdaBoost, Gradient Boosting fits the new model to the residual errors.

Put simply, Fit a model to the given Training set. Calculate the Residual Errors which become the new training instances. A new model is trained on these and so on. An addition of all the models is selected for making predictions.

The major difference between AdaBoost and Gradient Boosting Algorithm is how the two algorithms identify the shortcomings of weak learners (eg. decision trees). While the AdaBoost model identifies the shortcomings by using high weight data points, gradient boosting performs the same by using gradients in the loss function (y=ax+b+e ,e needs a special mention as it is the error term). The loss function is a measure indicating how good are model’s coefficients are at fitting the underlying data. A logical understanding of loss function would depend on what we are trying to optimise. For example, if we are trying to predict the sales prices by using a regression, then the loss function would be based off the error between true and predicted house prices. Similarly, if our goal is to classify credit defaults, then the loss function would be a measure of how good our predictive model is at classifying bad loans. One of the biggest motivations of using gradient boosting is that it allows one to optimise a user specified cost function, instead of a loss function that usually offers less control and does not essentially correspond with real world applications.

**4.3) RISK ANALYSIS AND MITIGATION**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **RISK ID** | **DESCRIPTION OF RISK** | **RISK AREA** | **PROBABILITY** | **IMPACT** | **RISK SELECTED FOR MITIGATION** | **MITIGATION PLAN** | **RE**  **(P\*I)** |
| 1 | Jupyter Notebook Lag and Shutdown due to large dataset | Systemand  software  failure | H | H | YES | Worked on Google Colaboratory which is a cloud based Resource providing software | H |
| 2 | Noisy Data | Data-Set | H | H | YES | Cleaning of data | H |
| 3 | For more accurate outputs large data set is needed | Maintainability | M | M | YES | Worked on  a system  with large computation power using a dataset of larger time period | M |
| 4 | Higher training time for algorithms like SVM and Ensemble | Performance | M | M | YES | Worked with GPU of Google Collab | M |
| 5 | If the Sequence or flow of events is interrupted | Coding and Implementation | L | H | YES | Completeness and validity of our cod | M |

**CHAPTER-5**

**TESTING**

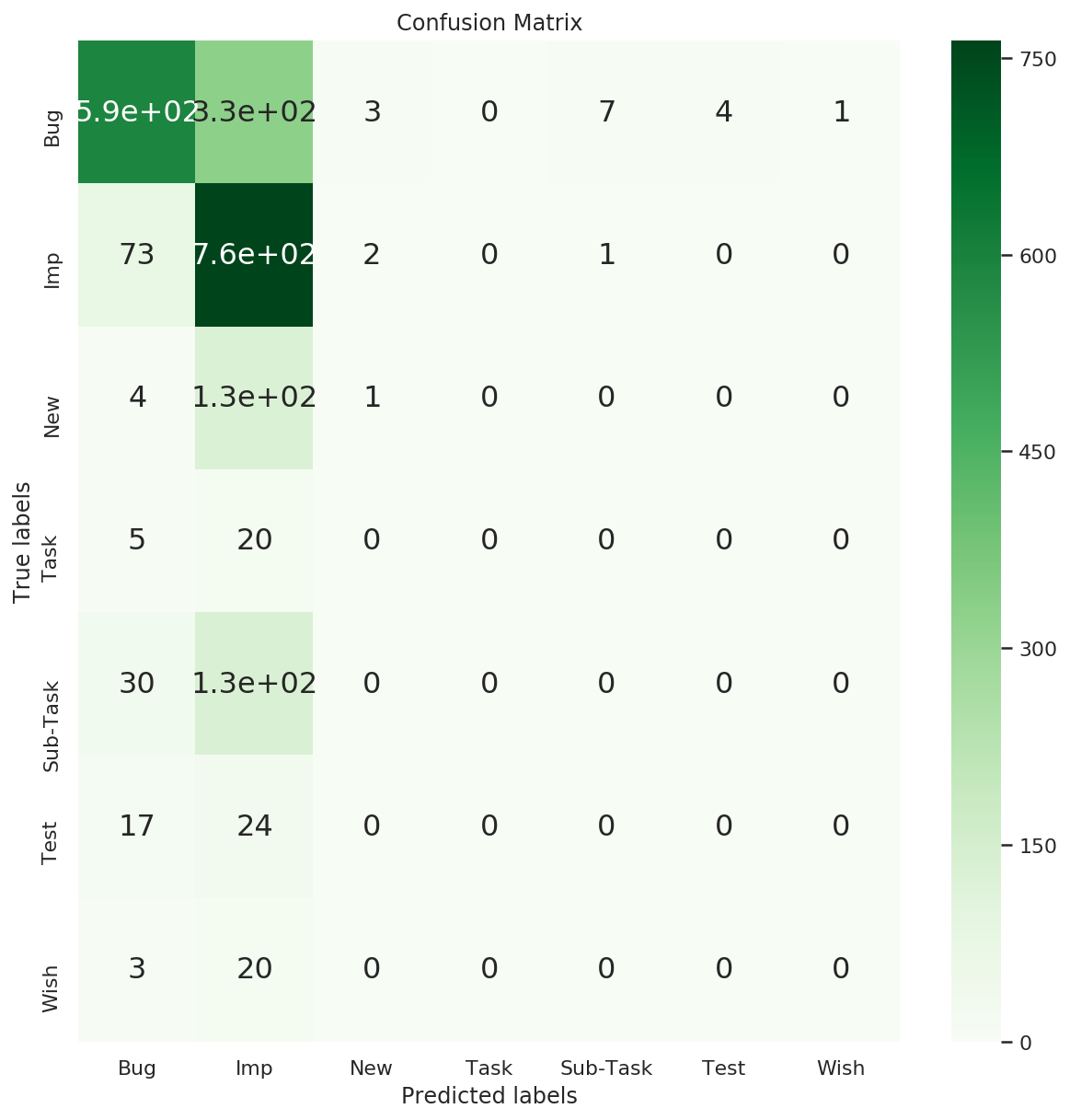
* 1. **TESTING PLAN**
  2. **COMPONENT DECOMPOSITION AND TYPE OF TESTING REQUIRED**
  3. **TEST CASES**
  4. **LIMITATIONS**

The original dataset contained around 50 columns along with the comments made by the users which is very difficult to handle and process the useful information from the original Dataset. So, we Selected 10 useful columns which were required for the empirical study and NLP based analysis on summary and description. But in doing this so does the accuracy decreases. The absence of large amount of quality data is another challenge faced during this project completion. Most of the entries in the Summary title are merely name of the issues not providing any descriptive features for the particular classes. The third limitation was unbalanced dataset was used in this project which affected the accuracy of the models.

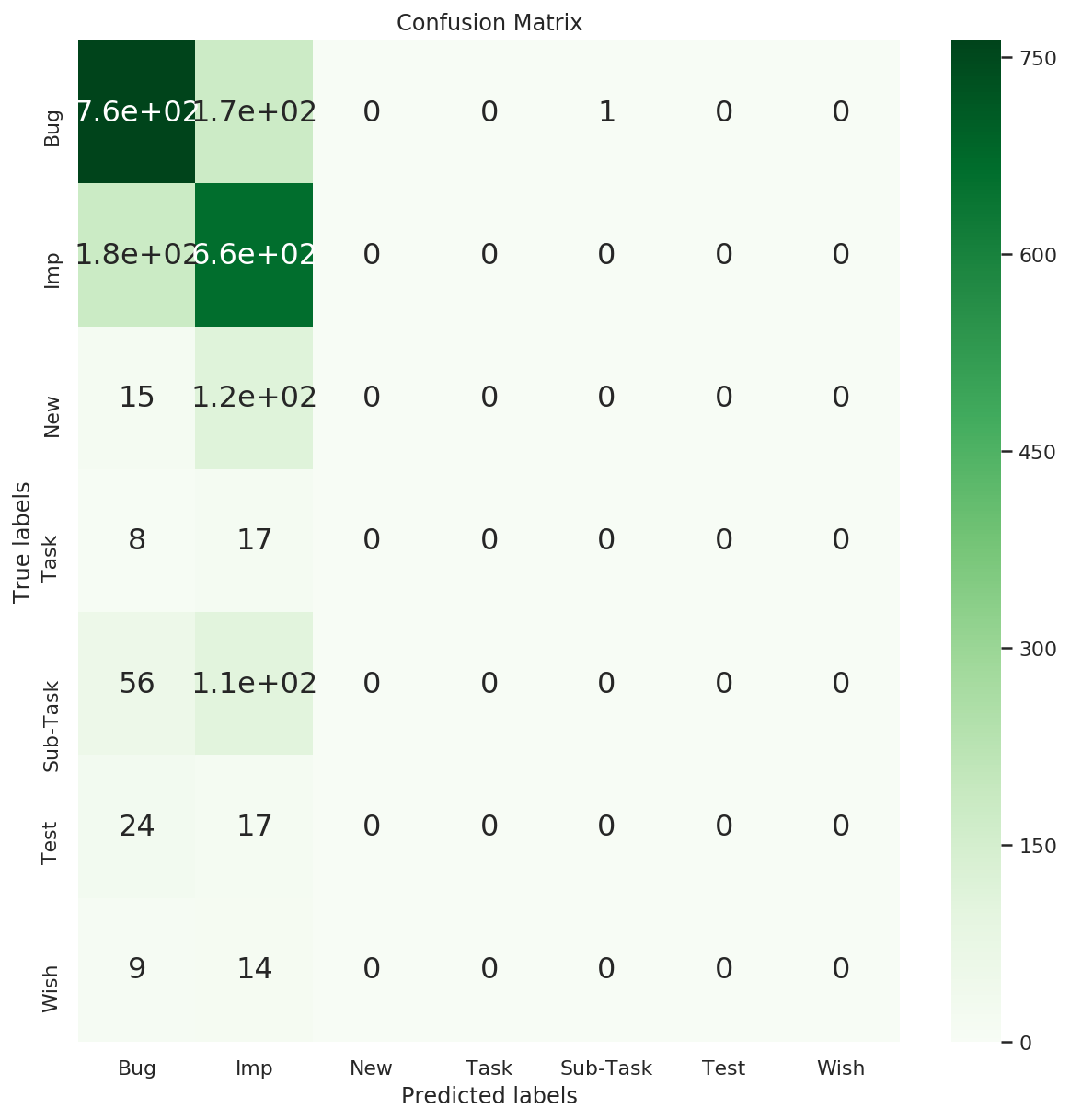
**CHAPTER-6**

**FINDING, CONCLUSION, FUTURE WORK**

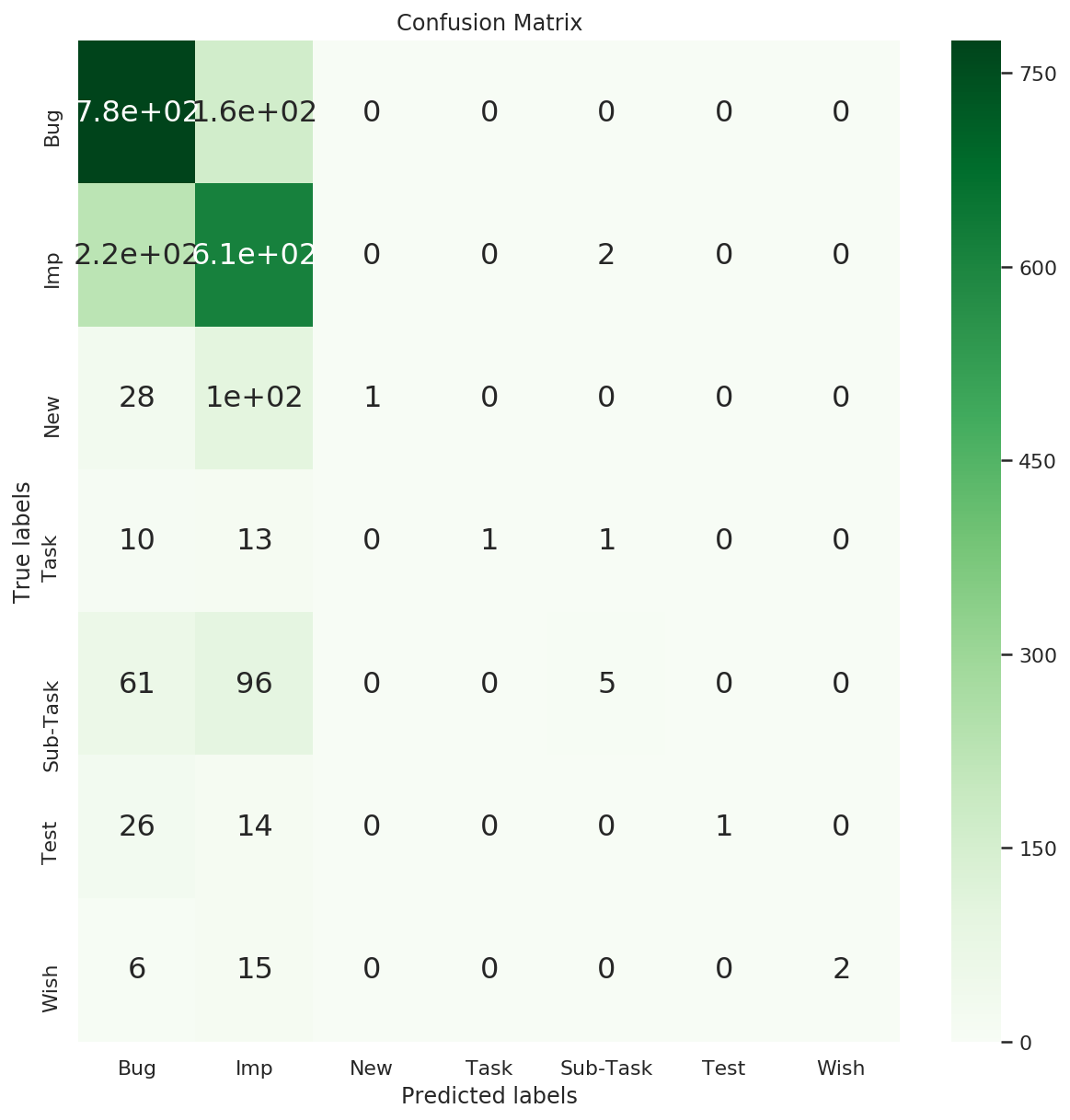
* 1. **FINDING**



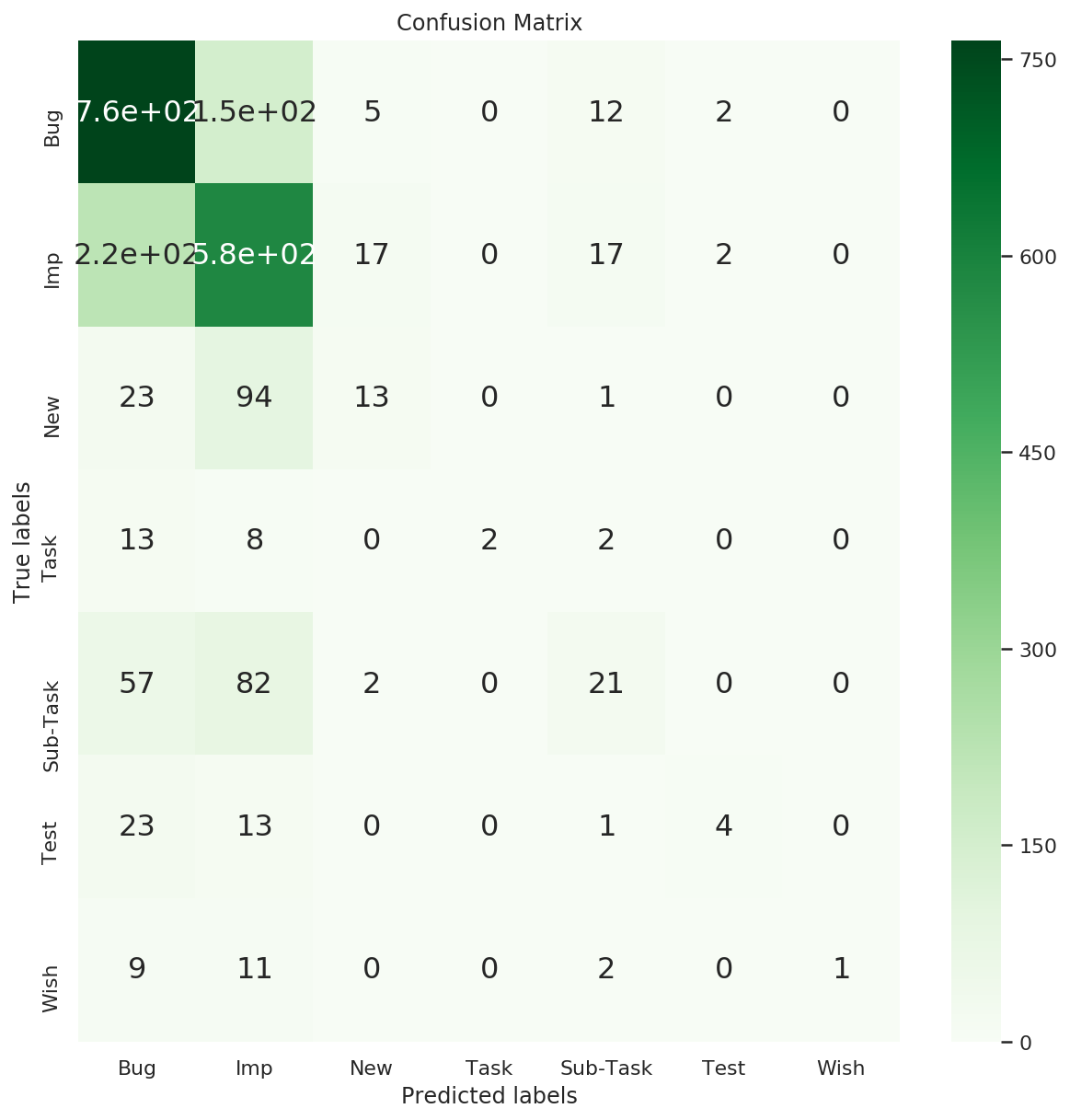
**Fig-10: Bernoulli Naïve Baye’s**



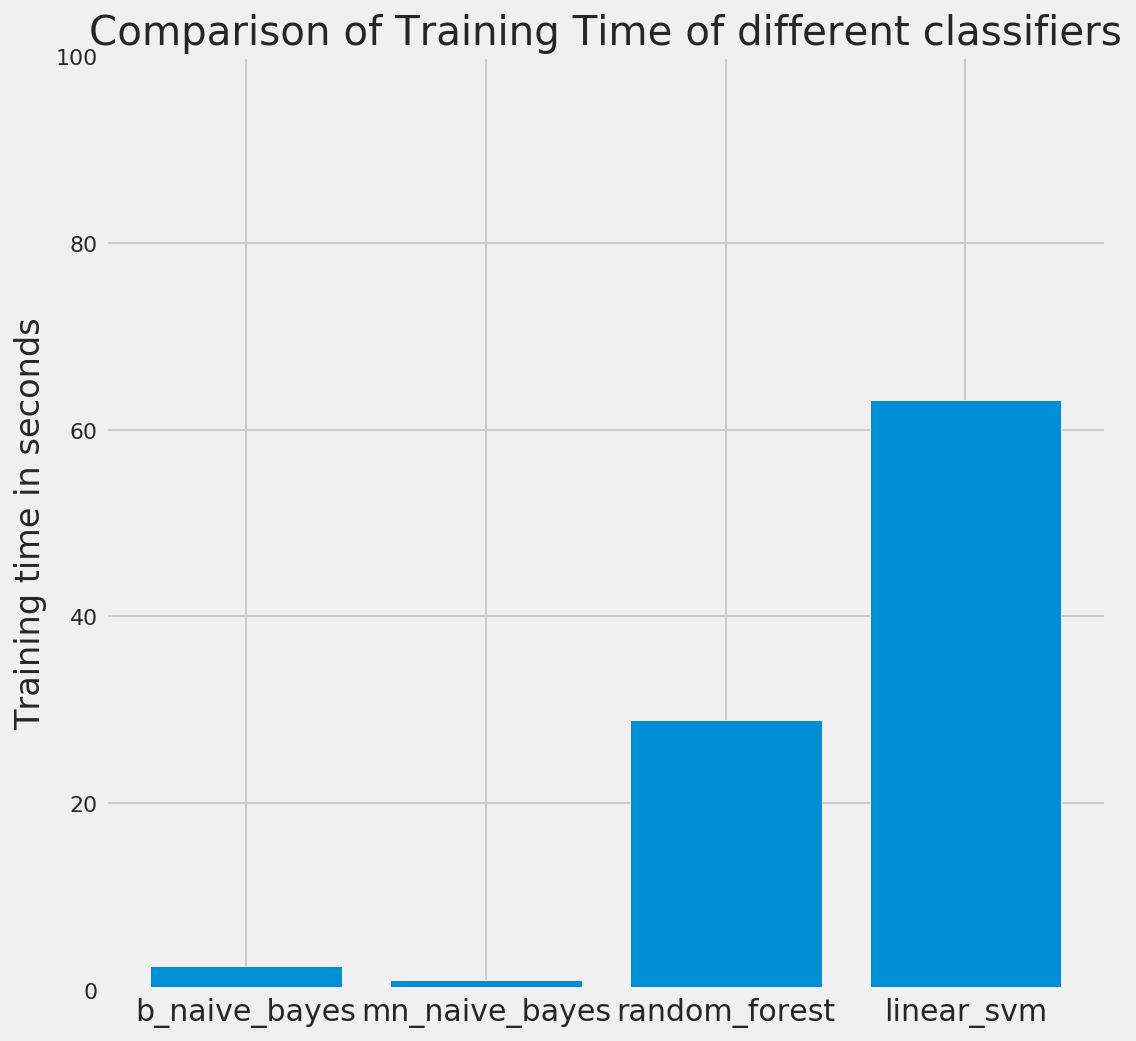
**Fig -11: Multinomial Naïve Bayes**

****

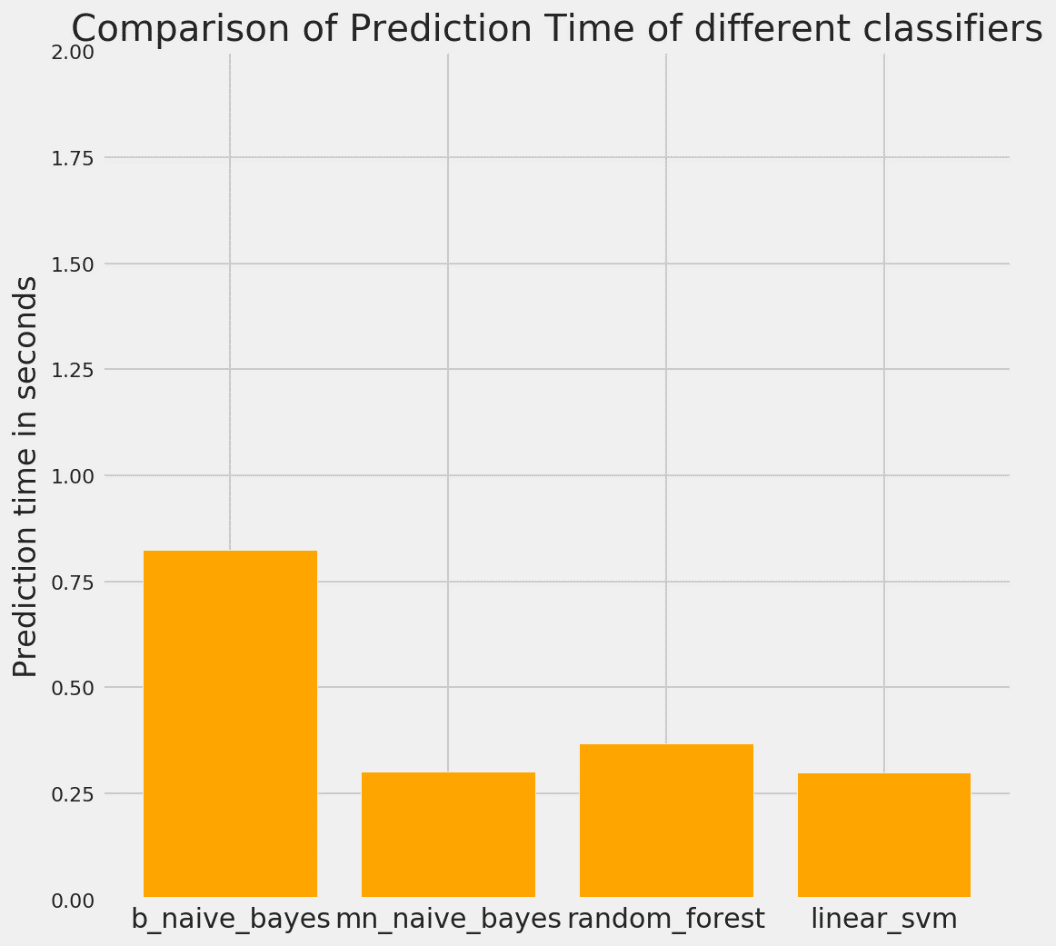
**Fig -12: Random Forest**

****

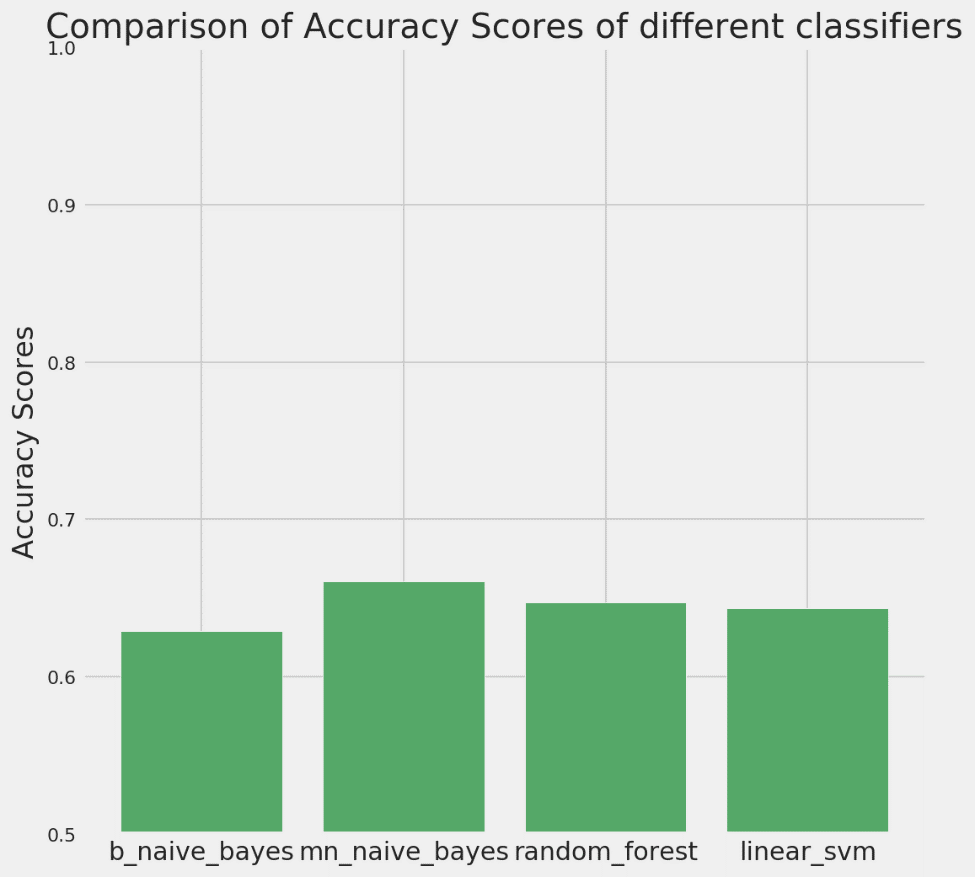
**Fig -13: SVM**

****

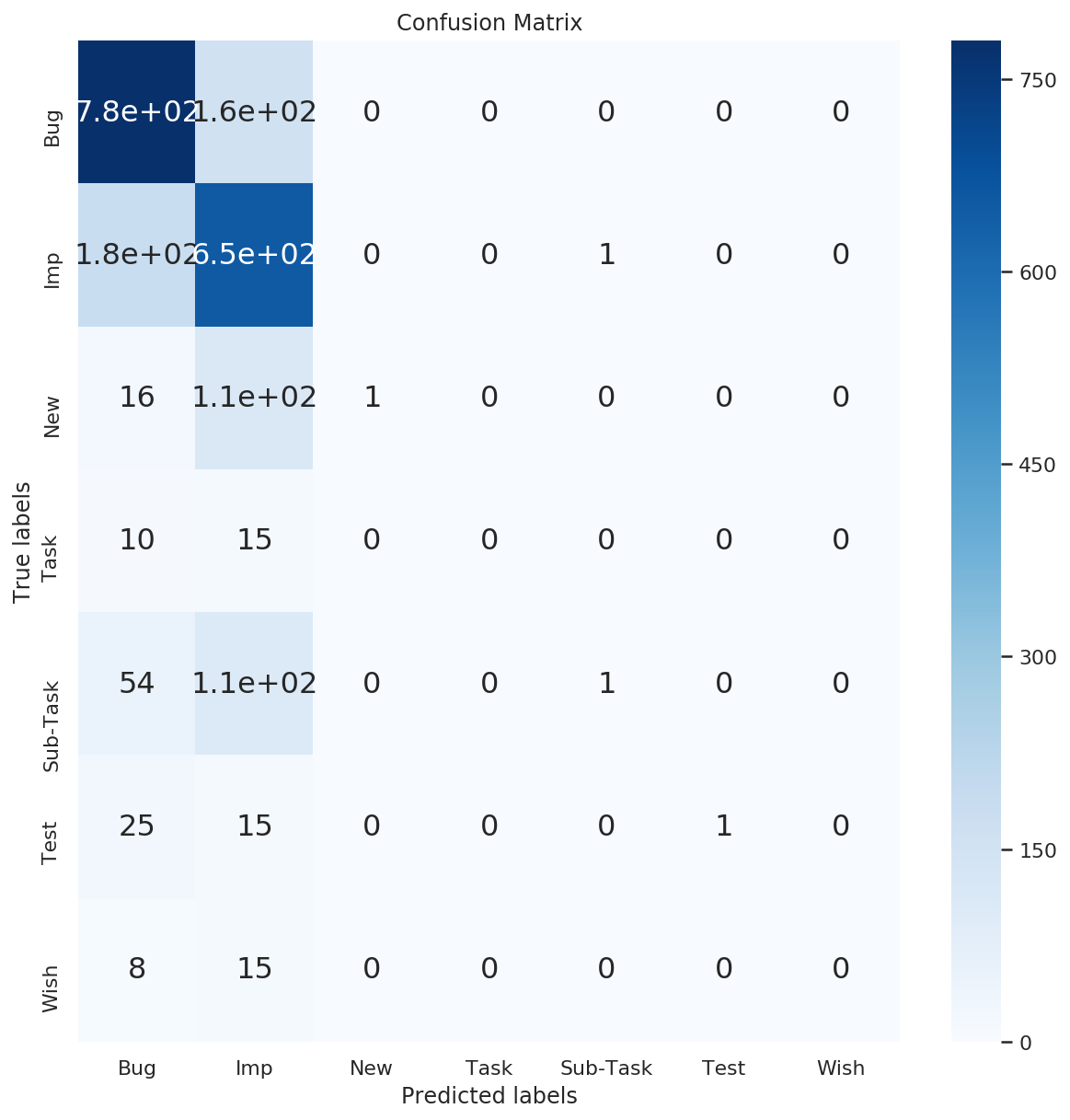
**Fig -14**

****

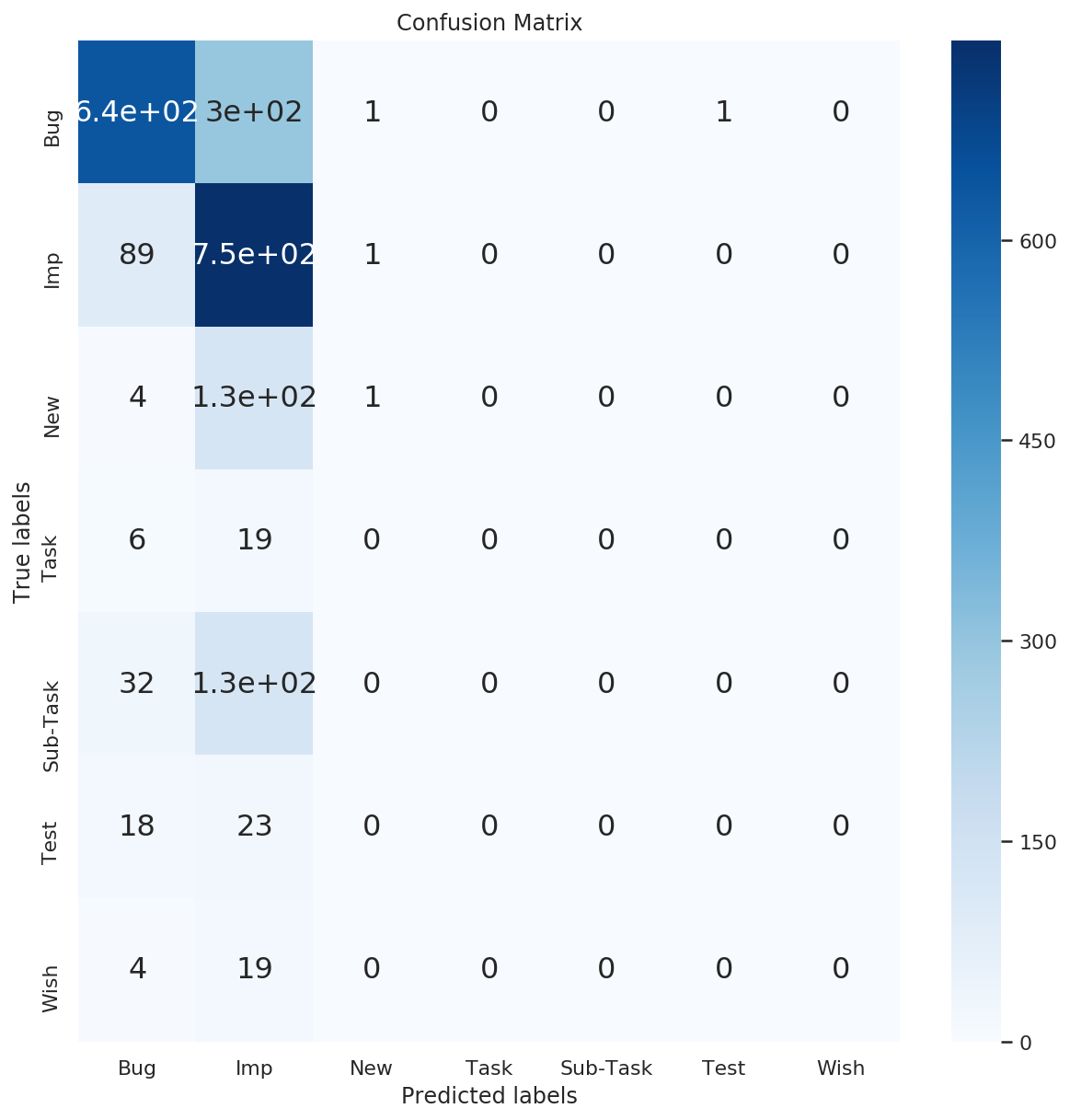
**Fig -15**

****

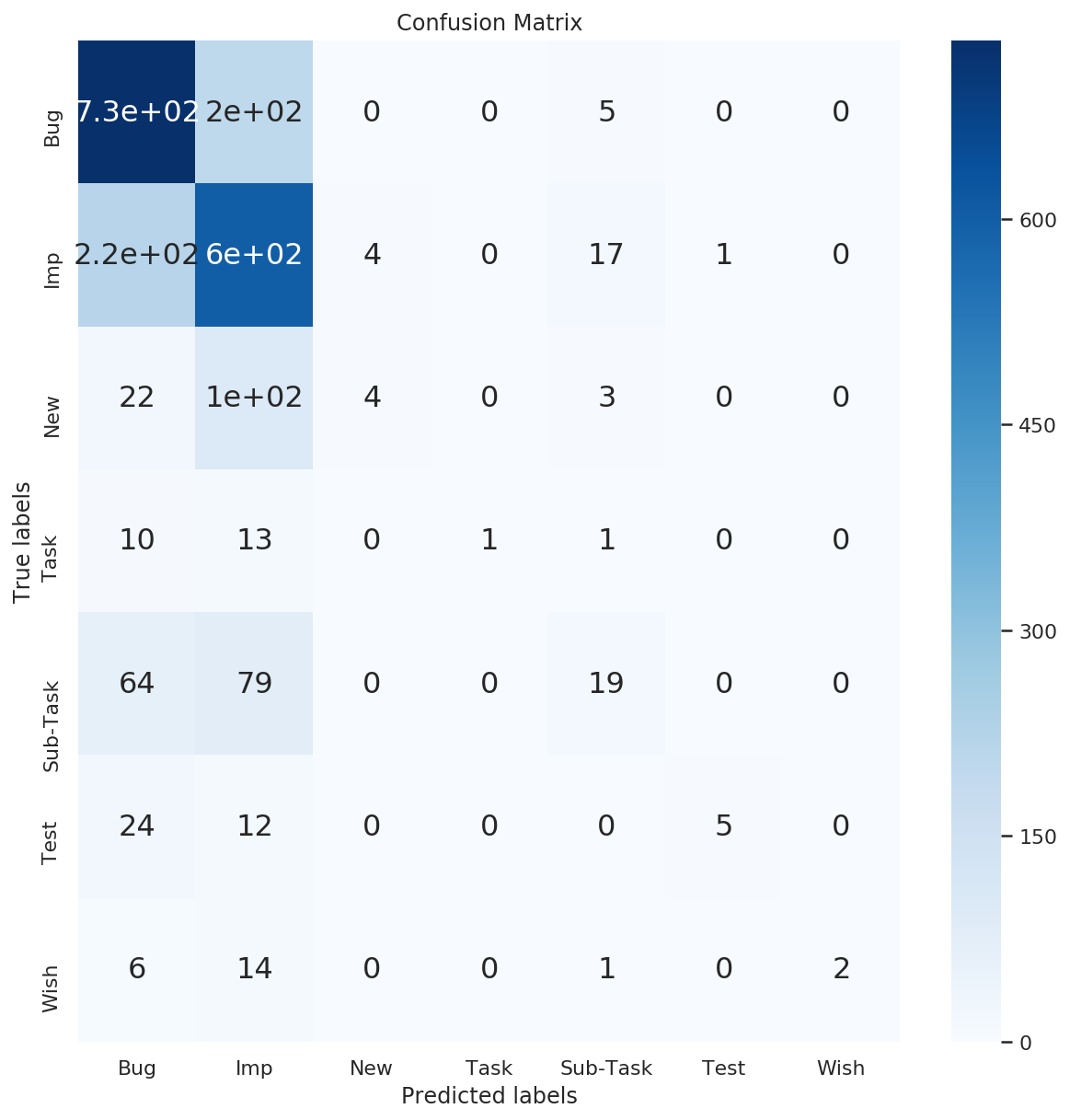
**Fig -16**

****

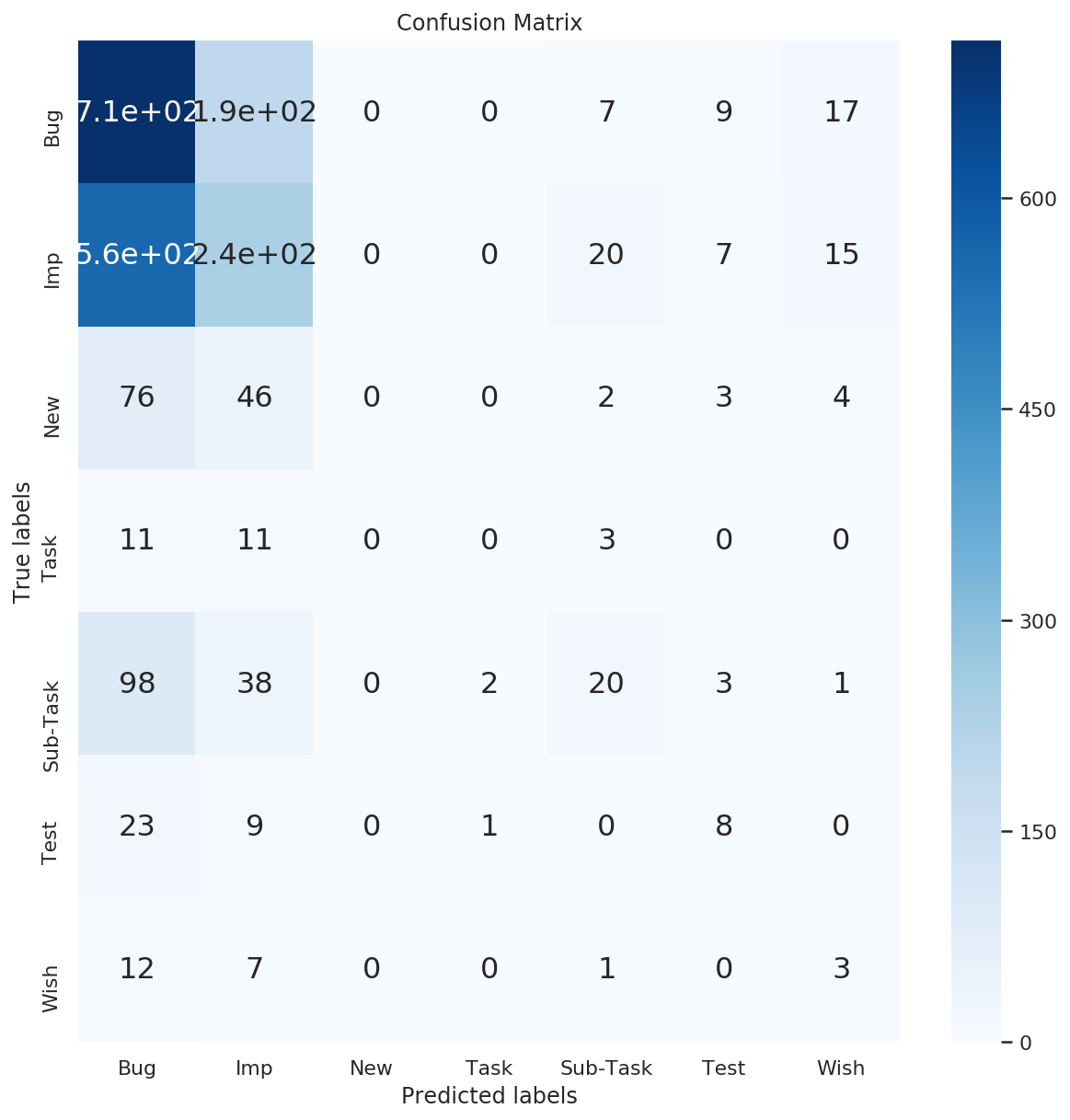
**Fig -17 Ensemble (Hard)**

****

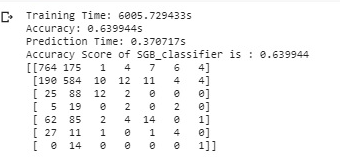
**Fig -18 Ensemble (Soft)**

****

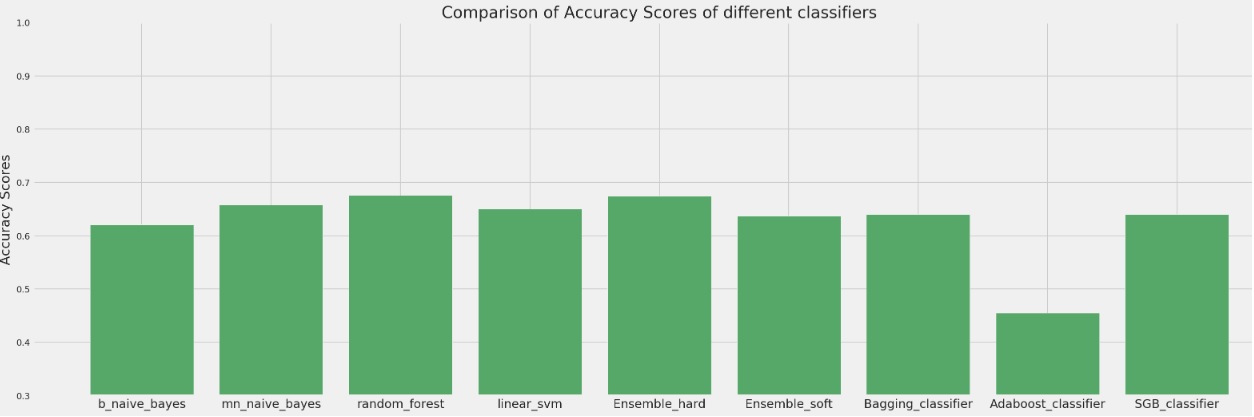
**Fig -19: Bagging**

****

**Fig -20: Ada-Boost**

****

**Fig -21: Stochastic Gradient Boosting**

****

**Fig- 22 : Overall Comparison**

**6.2) CONCLUSION**  
In this project we have used text data from the various bug reports by creating a dataset of those bug reports using the JIRA issue tracking system. The JIRA issue tracking system enables us to download a limit of 1000 bug reports and we have done it 8 times to create our dataset of 8000 bug reports. After data pre-processing the 8000 bug reports got reduced to 7191. We applied various machine learning algorithms like Naïve Bayes , Random Forest , SVM etc out of which we got our best performance in terms of accuracy in Random Forest, but its larger training time is somewhat degrading. So we experimented on the Ensemble approach and found out that majority voting algorithm actually performs the best but having the same downside as the Random Forest algorithm i.e. relatively larger training time. Bagging classifier andNaïve Bayes classifier performance is somewhat similar but Naïve Bayes offer a much lesser training time in comparison to Bagging, reducing our purpose to use bagged trees upon this dataset. Surprisingly Boosting methods like AdaBoost which was supposed to increase the accuracy of weak classifiers performed poorly upon our dataset.

We have been able to achieve an accuracy of net 67.86% on the testing dataset. The reason for the low accuracy in our opinion is the presence of high imbalance between the classes in our dataset. Had our Dataset been better our models of ensemble would have definitely proven to be more effective in comparison to single classification models.

**6.3) FUTURE WORK**

Below mentioned are some other factors and methodologies which could be considered while doing this analysis:

* Functional programming can be used,a general function could be formed which can be used to pass classifiers and to derive the results.
* Instead of using just machine learning models, we could also use Neural networks (MLPs) and other deep learning network to achieve better results.
* Using latent factorization methods like Non-negative matrix factorization, we can find higher level features that can then be used during classification.
* We can further augment the feature extraction process by assigning different weights to the text in different positions e.g. assigning more weight to the text in title and the text at the starting sections of the body. This could be used to explore if it improves the results or not.
* We can also explore forming n-gram features to see if those generate any better results or not.
* Similarly, we can use advanced methodology like word2vec and to find out words that occur together and can use them in the features extraction process as well.
* Another method to validate the results of classifiers can be Area under the curve (AUC) of ROC curve. Using One-Vs-All classification, we can form AUC to further assess the performance of our classifiers.

**(V)**

**REFERENCES**

1. Lal, S., &Sureka, A. (2012, December). Comparison of seven bug report types: A case-study of google chrome browser project. In *2012 19th Asia-Pacific Software Engineering Conference* (Vol. 1, pp. 517-526). IEEE.
2. Catolino, G., Palomba, F., Zaidman, A., &Ferrucci, F. (2019). Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, *152*, 165-181.
3. Pandey, N., Sanyal, D. K., Hudait, A., & Sen, A. (2017). Automated classification of software issue reports using machine learning techniques: an empirical study. *Innovations in Systems and Software Engineering*, *13*(4), 279-297.
4. Zaman, S., Adams, B., & Hassan, A. E. (2011, May). Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories* (pp. 93-102). ACM.
5. Herzig, K., Just, S., & Zeller, A. (2013, May). It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 international conference on software engineering* (pp. 392-401). IEEE Press.
6. Javed, M. Y., &Mohsin, H. (2012, July). An automated approach for software bug classification. In *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems* (pp. 414-419). IEEE.
7. Bhattacharya, P., &Neamtiu, I. (2010, September). Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *2010 IEEE International Conference on Software Maintenance* (pp. 1-10). IEEE.
8. Mani, S., Sankaran, A., &Aralikatte, R. (2019, January). Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data* (pp. 171-179). ACM.
9. Nagwani, N. K., Verma, S., & Mehta, K. K. (2013, November). Generating taxonomic terms for software bug classification by utilizing topic models based on Latent Dirichlet Allocation. In *2013 Eleventh International Conference on ICT and Knowledge Engineering* (pp. 1-5). IEEE.
10. Zhou, C., Li, B., Sun, X., &Guo, H. (2018, May). Recognizing software bug-specific named entity in software bug repository. In *Proceedings of the 26th Conference on Program Comprehension* (pp. 108-119). ACM.
11. Tian, Y., Sun, C., & Lo, D. (2012, March). Improved duplicate bug report identification. In *2012 16th European Conference on Software Maintenance and Reengineering* (pp. 385-390). IEEE.
12. Sureka, A., &Jalote, P. (2010, November). Detecting duplicate bug report using character n-gram-based features. In *2010 Asia Pacific Software Engineering Conference* (pp. 366-374). IEEE.
13. Maalej, W., & Nabil, H. (2015, August). Bug report, feature request, or simply praise? on automatically classifying app reviews. In *2015 IEEE 23rd international requirements engineering conference (RE)* (pp. 116-125). IEEE.
14. Murphy, G., &Cubranic, D. (2004, June). Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*
15. Anvik, J., Hiew, L., & Murphy, G. C. (2006, May). Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering* (pp. 361-370). ACM.