AUTOMATIC BUG TRIAGING
A MACHINE LEARNING APPROACH

A PROJECT REPORT

Submitted By

Saagarikha S. **312211104086**

Susindaran E. **312211104111**

Venugopal C G. **312211104121**

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SSN COLLEGE OF ENGINEERING

KALAVAKKAM 603110

ANNA UNIVERSITY :: CHENNAI - 600025

April 2015

ANNA UNIVERSITY : CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report titled **"Automatic Bug Triaging A Machine Learning Approach"** is the *bonafide* work of "**Saagarikha. S (312211104086)**, **Susindaran. E (312211104111)**, and **Venugopal. C. G (312211104121)**" who carried out the project work under my supervision.

Dr. Chitra Babu
Head of the Department
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Dr. R. S. Milton
Supervisor
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on. . . . . .

Internal Examiner **External Examiner**

ACKNOWLEDGEMENTS

We thank GOD, the almighty for giving me strength and knowledge to do this project.

We would like to thank and deep sense of gratitude to my guide **Dr. R. S. Milton**, Professor, Department of Computer Science and Engineering, for his valuable advice and suggestions as well as his continued guidance, patience and support that helped me to shape and refine our work.

Our sincere thanks to **Dr. CHITRA BABU**, Professor and Head of the Department of Computer Science and Engineering, for her words of advice and encouragement and we would like to thank our project Coordinator **Dr. S. SHEERAZUDDIN**, Professor, Department of Computer Science and Engineering for his valuable suggestions throughout this first phase of project.

We express our deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. We also express our appreciation to our **Dr. S. SALIVAHANAN**, Principal, for all the help he has rendered during this course of study.

We would like to extend our sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of our project work. Finally, we would like to thank our parents and friends for their patience, cooperation and moral support throughout our life.

Saagarikha S. **Susindaran E.** **Venugopal C G.**

ABSTRACT

For popular software systems, the number of daily submitted bug reports is high. Triaging these incoming reports is a time consuming task. Part of the bug triage is the assignment of a report to a developer with the appropriate expertise. In this paper, we present an approach to automatically suggest developers who have the appropriate expertise for handling a bug report, based on the identified component obtained from the short description of the bug report. Our work is the first to examine the impact of multiple machine learning dimensions (classifiers and training history) along with the ranked list of developers for prediction accuracy in bug assignment. We validate our approach on Eclipse covering 2,868,000 bug reports consisting of 253 components. We demonstrate that our techniques can achieve upto 80.05% prediction accuracy in bug assignment and significantly reduce the aberrant assignment of bugs. We compared the prediction time for our dataset using various algorithms such as Naive Bayes Text Classifier, Multinomial Naive Bayes and Linear SVM. We arrived at a conclusion that SVM provides higher accuracy and less learning time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Motivation

To bug is human, to debug is divine. Software evolution has high associated costs and effort. A survey by the National Institute of Standards and Technology estimated that the annual cost of software bugs is about $59.5 billion. Some software maintenance studies indicate that maintenance costs are atleast 50% and sometimes more than 90%, of the total costs associated with a software product. These surveys suggest that making the bug fixing process more efficient would reduce evolution effort and lower software production costs.

Most software projects use bug trackers to organize the bug fixing process and facilitate application maintenance. For instance, Bugzilla is a popular bug tracker used by many large projects, such as Mozilla, Eclipse, KDE, and Gnome. These applications receive hundreds of bug reports a day; ideally, each bug gets assigned to a developer who can fix it in the least amount of time. This process of assigning bugs, known as bug assignment, is complicated by several factors: if done manually, assignment is labour-intensive, time-consuming and fault-prone; moreover, for open source projects, it is difficult to keep track of active developers and their expertise. Identifying the right developer for fixing a new bug is further aggravated by growth, e.g., as projects add more components, modules, developers and testers, the number of bug reports submitted daily increases, and manually recommending developers based on their expertise becomes difficult.

Reports indicate that, on average, the Eclipse project takes about 40 days to assign a bug to the first developer, and then it takes an additional 100 days or more to

reassign the bug to the second developer. Similarly, in the Mozilla project, on average, it takes 180 days for the first assignment and then an additional 250 days if the first assigned developer is unable to fix it. These numbers indicate that the lack of effective, automatic assignment and toss reduction techniques results in considerably high effort associated with bug resolution. Reassigning a bug to another developer if the previous assignee is unable to resolve it is known as Bug Tossing. It can be inferred from the dataset of Eclipse that almost 90% of all ”Fixed” bugs have been tossed at least once.

CHAPTER 2

Design

2.1 Module Diagram

Module Diagram illustrates step-by-step process carried out for accurately assigning the bug report to the developer. Initially, the bug reports of training data set is text processed. The term selection methods reduce the sparseness of the bug report. The classifier learns from the refined bug report and its used to accurately identify the developer. When the new bug report is given the system predicts the accurate developer based on the learned classifier.

FIGURE 2.1: MODULE DIAGRAM

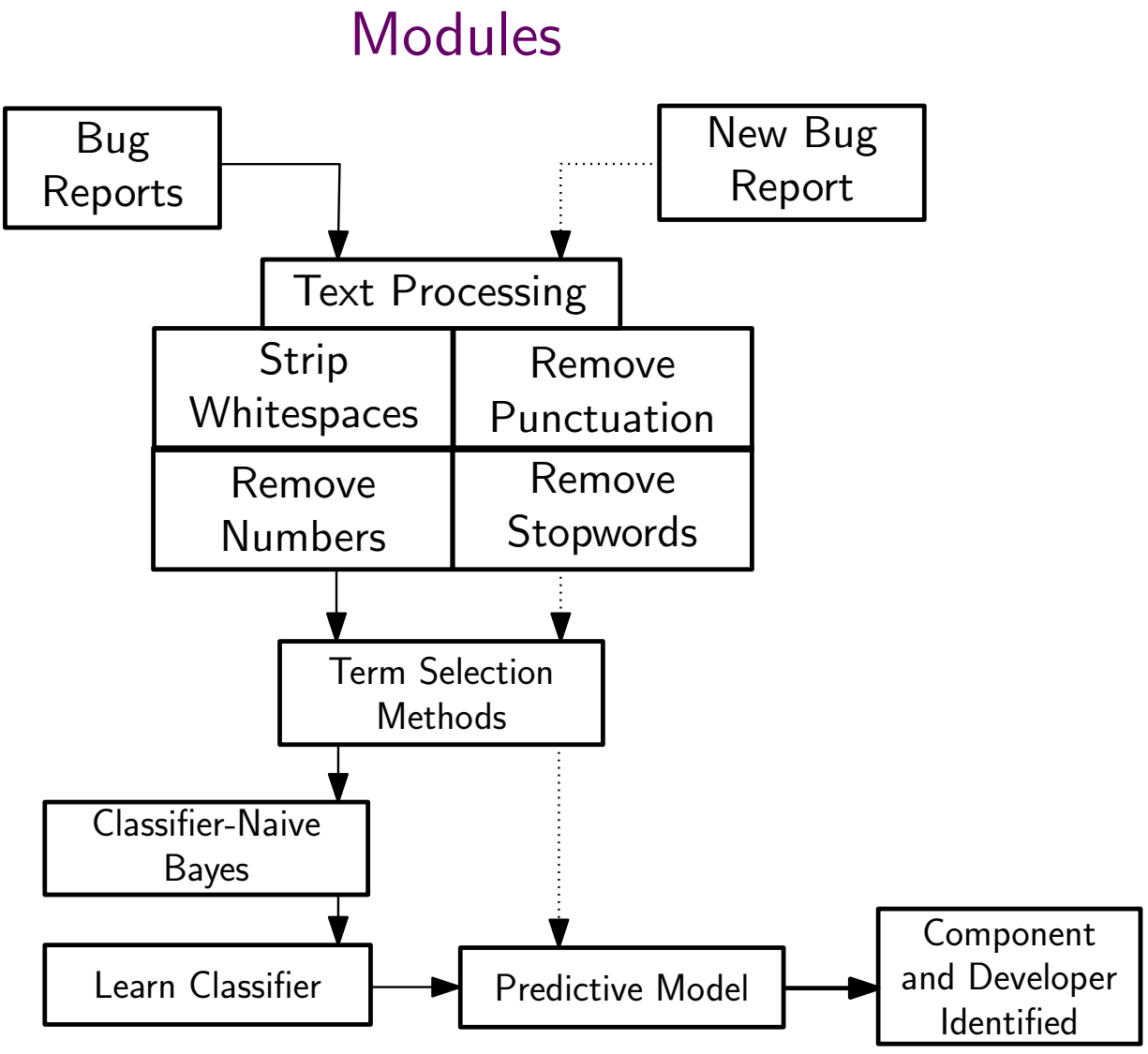
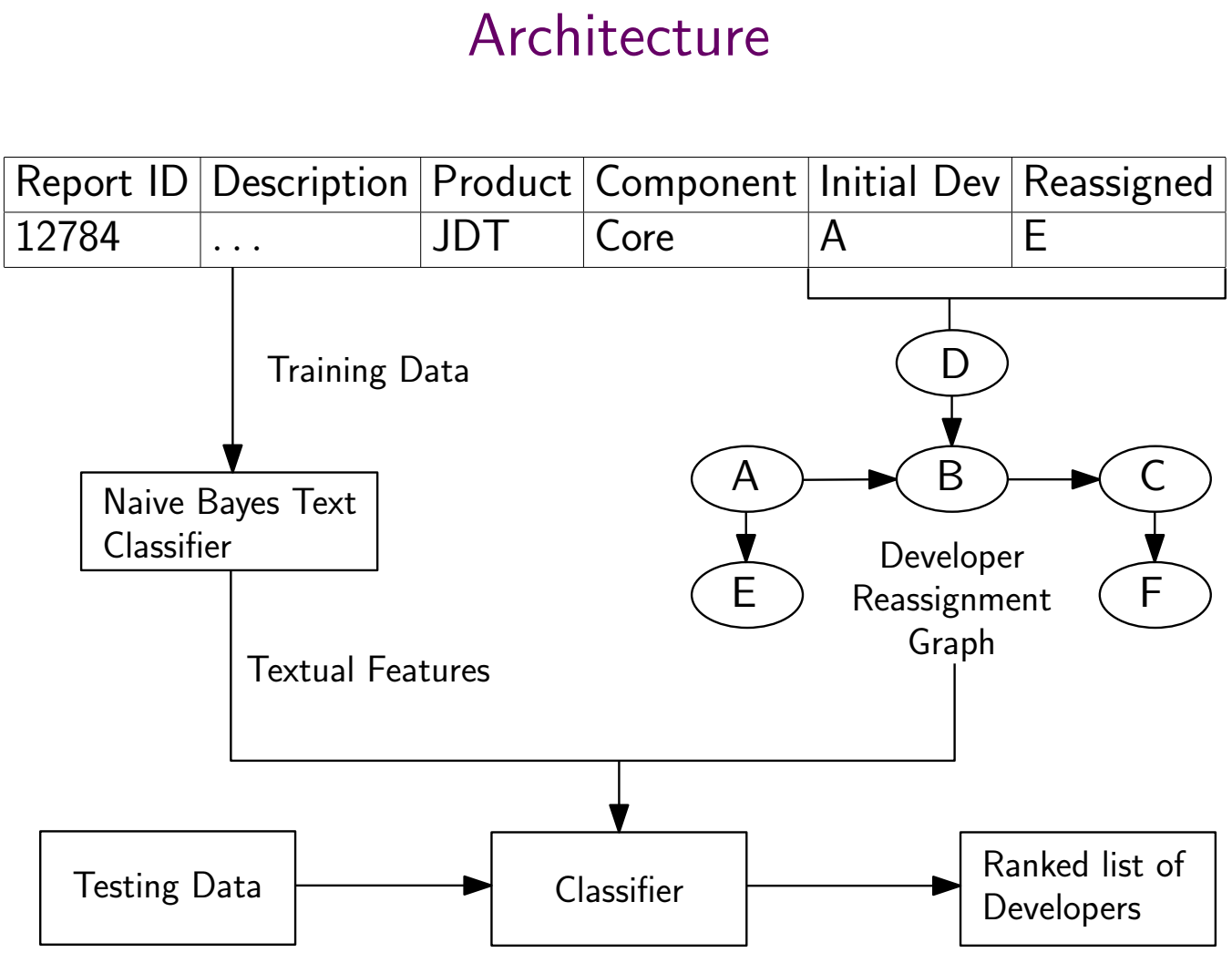


FIGURE 2.2: ARCHITECTURE DIAGRAM



CHAPTER 3

Literature Survey

Cubranic et al. [1] were the first to propose the idea of using text classification methods (similar to methods used in machine learning) to semi-automate the process of bug assignment. They used keywords extracted from the title and description of the bug report, as well as developer IDs as attributes, and trained a Naive Bayes classifier. When presented with new bug reports, the classifier suggests one or more potential developers for fixing the bug. Their method used bug reports for Eclipse for training, and reported a prediction accuracy of up to 30%. While we use classification as a part of our approach, in addition, we employ incremental learning and tossing graphs to reach higher accuracy.

Anvik et al. [2] improved the machine learning approach proposed by Cubranic et al. by using filters when collecting training data: (1) filtering out bug reports labeled invalid, wontfix, or worksforme, (2) removing developers who no longer work on the project or do not contribute significantly, and (3) filtering developers who fixed less than 9 bugs. They used three classifiers, SVM, Naive Bayes and C4.5. They observed that SVM (Support Vector Machines) performs better than the other two classifiers and reported prediction accuracy of up to 64

Lin et al. [3] conducted machine learning-based bug assignment on a proprietary project, SoftPM. Their experiments were based on 2,576 bug reports. They report 77.64% average prediction accuracy when considering module ID (the module a bug belongs to) as an attribute for training the classifier; the accuracy drops to 63% when module ID is not used. Their finding is similar to our observation that

using product-component information for classifier training improves prediction accuracy.

Jeong et al. [4] introduced the idea of using bug tossing graphs to predict a set of suitable developers for fixing a bug. They used classifiers and tossing graphs (Markov-model based) to recommend potential developers.

Pamela bhattacharya et al.[5] proposed a technique for automated bug assignment using machine learning and tossing graphs. They used a classifier and a tossing graph to automatically assign a bug to a developer. Initially, they used a training data set of fixed bugs that contains information regarding the developers to whom it was assigned and the reassignment to other developers.

CHAPTER 4

Algorithms for Automatic Bug Triaging

The various algorithms for Automatic Bug Triaging depend on the dataset being used. The training dataset obtained after parsing, has been subjected to stop-word removal and stemming. Snowball Stemming algorithm is used for stop-word removal and stemming. Multinomial NB classifier is used for converting the data set to feature vectors.

The product, component and short description of each bug from the Training data set are parsed using a XSLT parser to obtain an unified text file. The report ID of each bug from the `assigned-to.xml` file is taken and the ”when” attribute of each update of the bug is taken and matched with the corresponding entries in `short-desc.xml`, `product.xml` and `component.xml` and extracted and output to a text file in a format suitable for text processing.

4.1 Snowball Stemming Algorithm

The above parsed data has to be processed to extract the keywords. Stop-word and special Characters removal is first performed using the Natural Language ToolKit

in python. Then the same Toolkit is used to perform stemming on the text after the stop-words are removed.

Algorithm 1: SNOWBALL STEMMING - Removes Stop-words and performs Stemming

Input: F: Text file containing the Training dataset
S: Set of stop-words for the language English

Output: FP : Text file containing the Training dataset after stop-word removal and Stemming

```
foreach line L in F do // For each line in the file
    foreach word W in L do // For each word in the line
        if W not in S then // If the word is not a stop-word
            STEM(W)
        end
        /* Performs Stop-word removal and Stemming on each W    */
    end
end
return FP
```

4.2 Naive Bayes Text Classifier Algorithm

Through supervised learning the system learns from the TDS(Training data set)- the keywords present in the description of the bug and the developer to whom it was assigned. It also gathers information about the reassignment of bugs.

The system uses Naive Bayes classifier to classify the new bugs reported, to calculate the probability of it being assigned to a developer. Naive Bayes is a

probabilistic technique that uses Bayes rule of conditional probability to determine the probability that an instance belongs to a certain class. Bayes rules states that, ”The probability of a class conditioned on an observation to the prior probability of the class times the probability of the observation conditioned on the class” and can be denoted as follows:

$$P(Class|Observation) = \frac{P(Observation|Class) * P(Class)}{P(Observation)}$$

For example, if the word ’concurrency’ occurs more frequently in the reports resolved by developer A than in the reports resolved by developer B, the classifier would predict A as a potential fixer for a new bug report containing the word ’concurrency’. Naive Bayes is so called because it makes the strong assumption that features are independent of each other, given the label (the developer who

resolved the bug).

Algorithm 2: NAIVE BAYES CLASSIFIER - Text classification

Input: T: Training corpus

B: New Bug Report

Output: d_j : The developer with the highest probability to whom the bug will be assigned.

From Training corpus, extract Vocabulary

foreach developer d_j in D **do** // Calculate P(d_j) terms

$reports_j \leftarrow$ *all bug reports in developer* d_j

$P(d_j) \leftarrow \frac{|reports_j|}{|total\ no.\ reports|}$

end

$Text_j \leftarrow$ *single text containing all* $reports_j$

foreach word w_k in *Vocabulary* **do** // Calculate P($w_k|d_j$) terms

$n_k \leftarrow no.\ of\ occurences\ of\ w_k\ in\ Text_j$

$P(w_k|d_j) \leftarrow \frac{n_k+const}{n+const|Vocabulary|}$

/* $const = 1$, Laplacian Smoothing constant */

end

return d_j *with highest probability*

4.3 Multinomial Naive Bayes Algorithm

Feature Vector : W=(w_1, w_2, w_3, ..., w_n)

$$log\ p(D_k|W) = log\ p(D_k) + \sum_{i=1}^{N} w_i\ .\ log\ p(w_i|D_k) \tag{4.1}$$

Prior : $\frac{|N_c|}{|N|}$ Conditional Probability : $\frac{count(w,D_k)+1}{count(D_k)+|V|}$

Algorithm 3: TRAIN MULTINOMIALNB - Text classification

Input: R: Training Corpus(List of bug reports)

C: List of developers

Output: V-vocabulary,prior and condprob

$V \leftarrow$ *extract Vocabulary*

$N \leftarrow$ *count bug Reports*

foreach developer d in D **do**

$N_c \leftarrow$ *count bug reports in developer d from R*

$prior(d) \leftarrow \frac{|N_c|}{|N|}$

$words_c \leftarrow$ *collect all words from all bug reports in developer c*

foreach word w in V **do**

$T_c \leftarrow$ *count occurences of word*$(w, words_c)$

$T'_c \leftarrow$ *countwords*(c)

$condprob(w|c) \leftarrow \frac{|T_c+1|}{T'_c+1}$

end

end

return V *, prior and condprob*

Algorithm 4: APPLY MULTINOMIALNB - Text classification

Input: C: List of Developers

V: Vocabulary

prior

condprob

R: Bug Report

Output: d: The developer with the highest probability to whom the bug maybe assigned to.

$W \leftarrow$ *extract words from Report R*

foreach developer d in D **do**

  $P(d|R) \leftarrow log(prior(d))$

  foreach word w in W **do**

    freq = count(w,R)

    $P(d|R)$ += freq * log(condprob($w|d$))

  end

end

return $argmax_d\ P(d|R)$

4.4 Tf- idf Weight Calculation

Term Frequency - Inverse Document Frequency(tf-idf) is a numerical statistic that is intended to reflect how important is a word to a document in a collection or corpus. This is to minimize the frequently occurring words in all the documents(bug reports) from reducing the efficiency in bug assignment.

$$tf(t,doc) = 0.5 + \frac{0.5 \times f(t,doc)}{\max\{f(w,doc) : w \in doc\}}$$

$$idf(t, DOC) = \log \frac{N}{1 + |\{doc \in DOC : t \in doc\}|}$$

$$tfidf(t, doc, DOC) = tf(t, doc) \times idf(t, DOC)$$

4.5 Linear Support Vector Machine

An Support Vector Machine (SVM) is a kind of large margin classifier. It is a vector space based Machine Learning method, where the goal is to find a decision boundary between two or more classes that is maximally far from any point in the training data. An Binary 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.

$$D = \{(x_i, y_i) \mid x_i \in R^p, y_i \in \{-1, 1\}\}_{i=1}^{n}$$

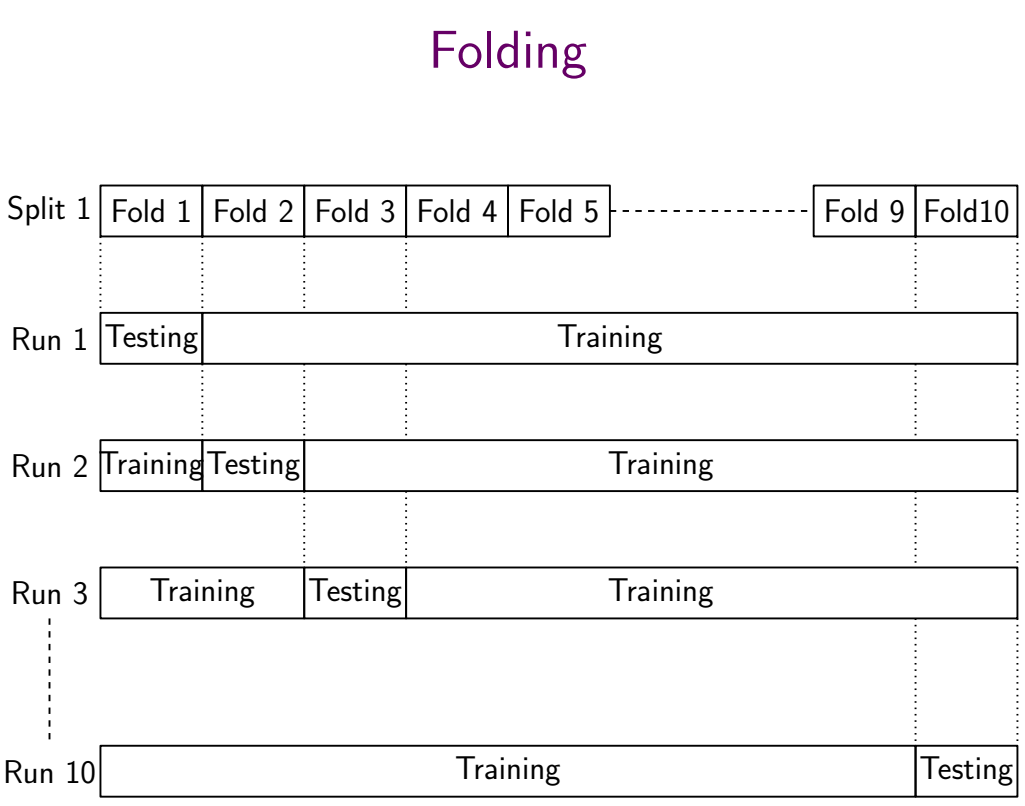
Multi-class SVM is implemented by reducing the single multi-class problem into multiple binary classification problems.(one-versus-all).

4.6 Folding

In folding based training and validation approach, also known as cross validation, the algorithm first collects all bug reports to be used for TDS (Training Data Set) sorts them in chronological order(based on the update time of the bug) and then divides them into n folds. In the first run, fold 1 is used to train the classifier and then to predict the VDS (Verified Data Set). In the second run, fold 2 bug reports

are added to TDS. In general, after validating the VDS from fold *n*, VDS is added to the TDS for validating fold *n+1*. For example, we chose *n=10* and carried out 9 iterations of the validation process using incremental learning.

FIGURE 4.1: FOLDING



4.7 Goal oriented tossing graph

When a bug is assigned to a developer for the first time and if she is unable to fix it, the bug is assigned(tossed) to another developer. Thus a bug is tossed from one developer to another until a developer is eventually able to fix it. Based on these tossing path, goal oriented tossing graphs were proposed. Tossing graphs are weighted directed graphs such that each node represents a developer, and each directed edge from D1 to D2 represents the fact that bug is assigned to developer D1 was tossed and eventually fixed by developer D2. The weight of the edge between two developers is the probability of a toss between them, based on bug tossing history. We denote a tossing event from developer D to $D_j asD \implies D_j$.

TABLE 4.1: GOAL ORIENTED TOSSING GRAPH

Tossing Graph [1]

Tossing paths
A→B→C→D
A→E→D→C
A→B→E→D
C→E→A→D
B→E→D→F

Developer who tossed the bug	Total tosses	Developer who fixed the bug					
		C		D		F	
		#	pr	#	pr	#	pr
A	4	1	0.25	3	0.75	0	0.00
B	3	0	0.5	2	0.67	1	0.33
C	2			2	1.00	0	0.00
D	2	1	0.50			1	0.50
E	4	1	0.25	2	0.50	1	0.25

4.8 Prediction Accuracy

If the first developer in our prediction list matches the actual developer who fixed the bug, we have a hit for the Top 1 developer count. Similarly, if the second developer in our prediction list matches the actual developer who fixed the bug, we have a hit for the Top 2 developer count. For example, if there are 100 bugs in the VDS and for 20 of those bugs the actual developer is the first developer in our prediction list, the prediction accuracy for Top 1 is 20%; similarly, if the actual developer is in our Top 2 for 60 bugs, the Top 2 prediction accuracy is 60%.

CHAPTER 5

Problem Definition and Proposed System

5.1 Problem Statement

Input: A bug report in natural language text submitted by the reporter briefing the problem.
Output: The component in which the bug may potentially be, and the developer or list of developers to whom it can be assigned to.

When the user finds a bug, he/she reports the bug through a bug tracker used by the Software. Since the description of the bug submitted by the user is a natural language text, Natural Language Processing is used to extract useful keywords from the bug report that would provide information about the bug that the user has encountered. The processing involves stop-word removal and stemming to extract useful keywords from the description of the bug report. These extracted keywords are used to identify the most probable defective component based on the dependencies that are previously learnt. Then based on the defective Component and Tossing History of the developers, a list of Developers will be informed of this bug to solve.

The list of Developers should be chosen in such a way that the probability of the bug getting reassigned must be minimum. After fixing the bug, the bug report is annotated/labelled with the developer and the component related to the bug. A dependency structure is formed over time for supervised learning from the fixed bugs.

5.2 Data Set

Dataset is a collection of fixed bug reports gathered from a open source software bug tracker tool containing necessary information about the components, developers and re-assignments. This is a categorized, classified, and semi-structured data. A bug report, generally a natural language text, submitted by the user is stored in the XML format by the bug tracker tool. Information contained in the dataset:

- severity: The severity denotes how soon the bug should be fixed.
- product: The particular software application the bug is related to.
- component: The relevant subsystem of the product for the reported bug.
- assigned to: The identifier of the developer to whom the bug was assigned to.
- short desc: Contains a natural language text embedded by the user.
- bug status: The status of the bug at every update. NEW, ASSIGNED, RESOLVED, VERIFIED, REOPENED.
- resolution: Tagging the bug report for maintenance. FIXED, REMIND, INVALID, WORKSFORME.

With these information the dependencies between the components, developers, and reassignment can be formed.

5.3 Sample Data set

```
<report id="122442">
<update>
     <when>1136131494</when>
       <what>jdt-core-inbox@eclipse.org</what>
    </update>
    <update>
       <when>1136182552</when>
       <what>frederic_fusier@fr.ibm.com</what>
</update>
</report>
```

A sample report in `assignedto.xml` data set. It specifies the developer to whom it was assigned to at every update along with the time of each update.

```
<report id="122442">
    <update>
      <when>1136131494</when>
       <what>API inconsistency with IJavaSearchConstants.IMPLEMENTORS and
    </update>
    <update>
       <when>1136182552</when>
       <what>[search] API inconsistency with IJavaSearchConstants.IMPLEME
</update>
</report>
```

The short description of the bug report submitted by the reporter.

```
<report id="122442">
    <update>
      <when>1136246754</when>
      <what>Core</what>
    </update>
    <update>
      <when>1136276291</when>
      <what>UI</what>
    </update>
  </report>
```

Gives information about the component involved at every update. The bug is trivially tracked using its report-id.

```
<xsl:choose>
        <xsl:when test="document('short_desc.xml')//short_desc/
report[@id=$w]/update[when=$x]/what!=''">
                <xsl:text>     </xsl:text>
                <xsl:value-of select="concat('what=',document
('short_desc.xml')//short_desc/report[@id=$w]/update[when=$x]
/what,'&#10;')">
                </xsl:value-of>
        </xsl:when>
</xsl:choose>
```

First the dataset in XML format is what we used but it has only around 10000 reports. To obtain higher efficiency, we used dataset which is in JSON format it has around 1,60,000 reports in a well structured manner. Training data set in JSON format compares the report-id and the update(”when”) of each report-id in

the respective files and merges the ”what” content present in short_desc(to get the bug report) component(to obtain the component) and assigned_to(the developer) to a single text file. This text file is pre-processed. The pre-processed file is converted to a feature-vector pair where the feature is the bug-report and the component it is present and the vector being the developer. The classifier learns from this feature-vector pair and predicts the accurate developer for incoming bug reports. Another feature-vector pair (component and developer) learned by the classifier is used for tossing graphs. The probability of developer solving the bug in particular component and his tossing to another developer are combined and the next probable developer who can fix the bug is determined.


```
{"assigned_to" : "123456": [{"who":86,
                                                    "what": Platform-UI-Inbox@eclipse.org, //developer/
                                                    "when":1023451345}, {"who":18,
                                                    "what": jhalhead@ca.ibm.com,
                                                    "when":10234513232} ],
                                "124323": [{"who":44,
                                                    "what": pde-ui-inbox@eclipse.org,
                                                    "when":101234245467}, {"who":11,
                                                    "what": Darin_Wright@oti.com,
                                                    "when":102345112342} ] }
{"short_desc" : "123456": [{"who":86,
                                                    "what": Proxy Dialog when invoking Content Assist , //
                                                    "when":1023451345}, {"who":18,
                                                    "what": Allow editing of non-project files,
                                                    "when":10234513232} ],
                                "124323": [{"who":44,
                                                    "what": content assist displays accessors,
                                                    "when":101234245467}, {"who":11,
                                                    "what": No refresh in package view when switching inte
                                                    "when":102345112342} ] }
{"component" : "123456": [{"who":86,
                                                    "what": Core, //Component/
                                                    "when":1023451345}, {"who":18,
                                                    "what": Text,
                                                    "when":10234513232} ],
                                "124323": [{"who":44,
                                                    "what": Text,
                                                    "when":101234245467}, {"who":11,
                                                    "what": UI,
```

"when":102345112342}] }

After parsing, the output text file contains data in the following format:

```
JDT, Type hierarchy returns type twice if executed on working copy layer,
JDT, Context Menu suggests I ca run applets that doesn’t exists, Debug
JDT, Identation broken when copying lines, Text
```

CHAPTER 6

Conclusion and Future Work

Machine learning and tossing graphs have proved to be promising for automating bug assignment. In this paper we lay the foundation for future work that uses machine learning techniques to improve automatic bug assignment by examining the impact of multiple machine learning dimensionslearning strategy, attributes, classifierson assignment accuracy.

- We used a broad range of text classifiers and found that, like many problems which use specific machine learning algorithms, we could able to select a specific classifier for the bug assignment problem.
- We validated our approach on two large, long-lived open-source projects; in the future, we plan to test how our current model generalizes to projects of different scale and lifespan. In particular we would like to find if the classifier preference should change as the project evolves and how source code familiarity of a developer could be used as an additional attribute for ranking developers.
- Similarly, when we assign tossing probabilities, we only consider the developer who could finally fix the bug. However, it is common that developers contribute partially to the final patch in various ways. For example, when a bug is assigned to a developer, he might provide insights and add notes to the bug report instead of actually fixing the bug; in fact, there are contributors who provide useful discussions about a bug in the

comment sections of a bug report who are never associated with the fixing process directly. These contributions are not considered in our ranking process,

CHAPTER 7

Result

7.1 Experimental Set-up

We used Mozilla and Eclipse bugs to measure the accuracy of our proposed algorithm. We analysed the entire life span of both applications. We divided our bug data sets into 10 folds and executed 9 iterations to cover all the folds. Data collection. We used the bug reports to collect four kinds of data: 1. Keywords: we collect keywords from the bug description and comments in the bug report. 2. Bug source: we retrieve the product and component the bug has been filed under from the bug report. 3. Temporal information: we collect information about when the bug has been reported and when it has been been fixed. 4. Developers assigned: we collect the list of developer IDs assigned to the bug

In our experiments, we varied the size of the test set, the size of the vocabulary. The below graph shows the classification accuracy as a function of the train/test set split, when the full vocabulary V of words found in bug reports is used.

As we can see, the algorithm correctly assigns just under 75% of the bugs, when 90% of the document corpus is used as training and 10% as the test set. The accuracy slowly declines to 65% as the test sets size is increased to 50the corpus.

The graph also shows the results when the vocabulary was created using stemming, which identifies most grammatical variations of a wordsuch as see, sees, seen, for

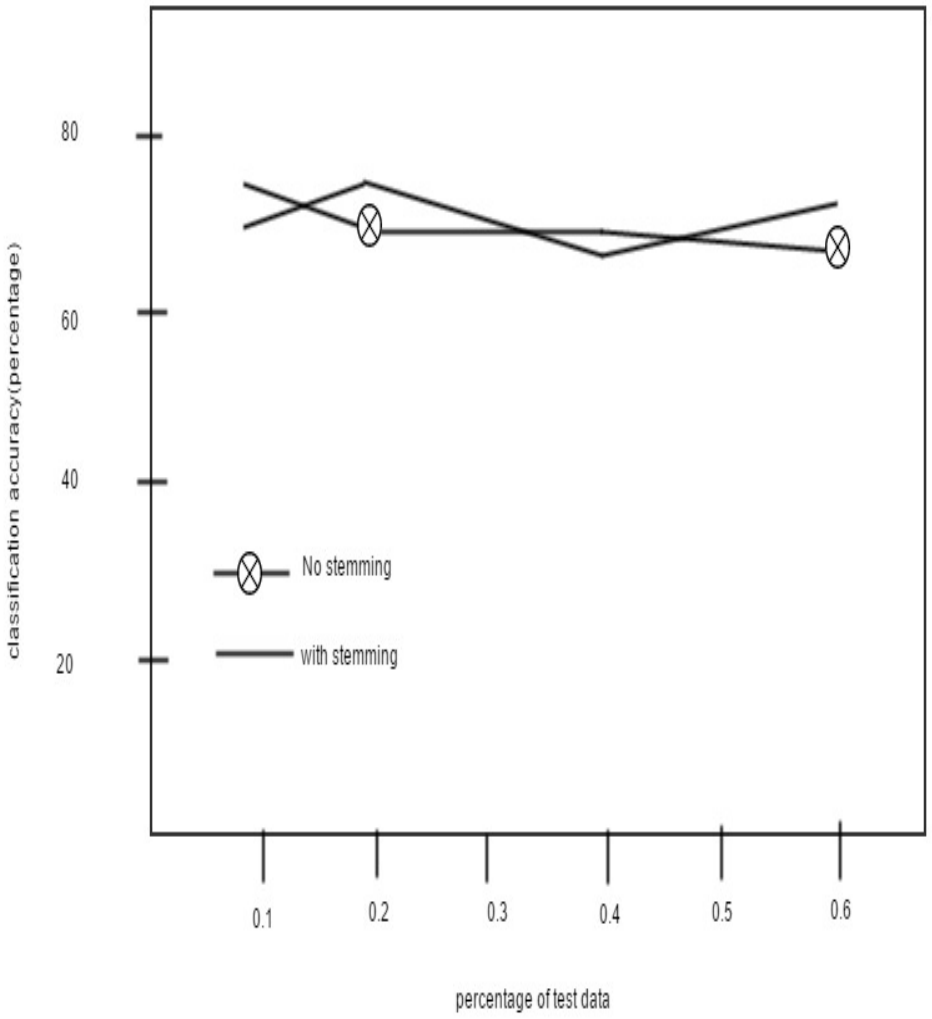


FIGURE 7.1: EXPERIMENTAL SET-UP

exampleand treats them as a single term. The results are virtually unchanged, and any differences between the two conditions are within about one standard deviation at each data point.

Finally with 10 fold cross validation an accuracy of nearly 78% is achieved and with linear SVM an accuracy of 80% is achieved.

REFERENCES

1. Anvik, J., Hiew, L., G. C. Murphy (2006), 'Who should fix this bug ?', In proc. of ICSE'06, pp. 361-370.

2. Bhattacharyaa, P., Neamtiua, I. (2010), 'Automated, Highly-Accurate, Bug Assignment Using Machine Learning and Tossing Graphs', In proc. of ICSM'10, pp.1-10.

3. Cubranic, D., Murphy, G.C., (2004), 'Automatic bug triage using text categorization', In proc. of SEKE'04 .

4. Jeong, G., Kim, S., Zimmermann, T., (2009), 'Improving Bug Triage with Bug Tossing Graphs', In proc. of FSE'09.

5. Lin, Z., Shu, F., Yang,Y., Hu,C., Wang, Q., (2009), 'An empirical study on bug assignment automation using Chinese bug data', In proc. of ESEM'09.

6. The Eclipse and Mozilla Defect Tracking Dataset(2013)
   Available From: $< https://github.com/ansymo/msr2013-bug\_dataset >$.